Table of Contents

Distributed Systems Engineering Lecture notes (MIT 6.824)

From: Distributed Systems Engineering notes (6.824, Spring 2015)

Lectures

Lecture notes from 6.824, taught by Prof. Robert T. Morris. These lecture notes are slightly modified from the ones posted on the 6.824 course website.

- Lecture 1: Introduction: distributed system definition, motivations, architecture, implementation, performance, fault-tolerance, consistency, MapReduce
- Lecture 2: Remote Procedure Calls (RPCs): RPC overview, marshalling, binding, threads, "at-least-once", "at-most-once", "exactly once", Go's RPC, thread synchronization
- Lecture 3: Fault tolerance: primary-backup replication, state transfer, "split-brain", Remus (NSDI 2008),
- Lecture 4: Flat datacenter storage: flat datacenter storage, bisection bandwidth, striping
- Lecture 5: Paxos: Paxos, consensus algorithms
  - Paxos algorithm description
- Lecture 6: Raft: Raft, a more understandable consensus algorithm
- Lecture 7: **Google Go** *guest lecture* by Russ Cox
- Lecture 8: Harp: distributed file system, "the UPS trick", witnesses
- Lecture 9: IVY: distributed shared memory, sequential consistency
- Lecture 10: TreadMarks: userspace distributed shared memory system, vector timestamps, release consistency (lazy/eager), false sharing, write amplification
- Lecture 11: Ficus: optimistic concurrency control, vector timestamps, conflict resolution
- Lecture 12: Bayou: disconnected operation, eventual consistency, Bayou
- Lecture 13: MapReduce: MapReduce, scalability, performance
- Lecture 14: **Spark** *guest lecture* by Matei Zaharia: Resilient Distributed Datasets, Spark
- Lecture 15: **Spanner** *guest lecture* by Wilson Hsieh, Google: Spanner, distributed database, clock skew
- Lecture 16: Memcache at Facebook: web app scalability, look-aside caches, Memcache
- Lecture 17: PNUTS Yahoo!: distributed key-value store, atomic writes
- Lecture 18: Dynamo: distributed key-value store, eventual consistency
- Lecture 19: **HubSpot** *guest lecture*
- Lecture 20: Two phase commit (2PC): two-phase commit, Argus
- Lecture 21: Optimistic concurrency control

- Lecture 22: Peer-to-peer, trackerless Bittorrent and DHTs: Chord, routing
- Lecture 23: Bitcoin: verifiable public ledgers, proof-of-work, double spending

Lectures form other years

- Practical Byzantine Fault Tolerance (PBFT): [2012], [2011], [2010], [2009], [2001], [PPT]

Papers

Papers we read in 6.824 (directory here):

1. MapReduce
2. Remus
3. Flat datacenter storage
4. Paxos
5. Raft
6. Harp
7. Shared virtual memory
8. TreadMarks
9. Ficus
10. Bayou
11. Spark
12. Spanner
13. Memcached at Facebook
14. PNUTS
15. Dynamo
16. Akamai
17. Argus, Guardians and actions
18. Kademlia
19. Bitcoin
20. AnalogicFS

Other papers:

1. Impossibility of Distributed Consensus with One Faulty Process
    - See page 5, slide 10 here to understand Lemma 1 (commutativity) faster
    - See this article here for an alternative explanation.
2. Practical Byzantine Fault Tolerance (PBFT)
    - See discussion here on PBFT.

Stumbled upon

1. A brief history of consensus, 2PC and transaction commit
2. Distributed systems theory for the distributed systems engineer
3. Distributed Systems: For fun and Profit
4. You can't choose CA out of CAP, or "You can't sacrifice partition tolerance"
5. Notes on distributed systems for young bloods
6. Paxos Explained From Scratch

Quizzes

Prep for quiz 1 here

6.824 2015 Lecture 1: Introduction

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Distributed systems

What is a distributed system?

- multiple networked cooperating computers
- *Example:* Internet E-Mail, Athena file server, Google MapReduce, Dropbox, etc.

Why distribute?

- to connect physically separate entities
- to achieve security via physical isolation
- to tolerate faults via replication at separate sites
- to increase performance via parallel CPUs/mem/disk/net

...but:

- complex, hard to debug
- new classes of problems, e.g. partial failure (did he accept my e-mail?)
- Leslie Lamport: *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*
- *Advice:* don't distribute if a central system will work

Why take this course?

- interesting -- hard problems, non-obvious solutions
- active research area -- lots of progress + big unsolved problems
- used by real systems -- unlike 10 years ago -- driven by the rise of big Web sites
- hands-on -- you'll build a real system in the labs

Course structure

See the course website.

Course components

- Lectures about big ideas, papers, labs
- Readings: research papers as case studies
  - please read papers before class
  - paper for today: MapReduce paper
  - each paper has a question for you to answer and one for you to ask (see web site)
  - submit question & answer before class, one or two paragraphs
- Mid-term quiz in class, and final exam
- Labs: build increasingly sophisticated fault-tolerant services
  - First lab is due on Monday
- Project: design and build a distributed system of your choice or the system we pose in the last month of the course
  - teams of two or three
  - project meetings with course staff
  - demo in last class meeting

Main topics

Example:

- a shared file system, so users can cooperate, like Dropbox
  - but this lecture isn't about dropbox specifically
  - just an example goal to get feel for distributed system problems
- lots of client computers

Architecture

- Choice of interfaces
  - Monolithic file server?
  - Block server(s) -> FS logic in clients?
  - Separate naming + file servers?
  - Separate FS + block servers?
- Single machine room or unified wide area system?
  - Wide-area dramatically more difficult.
- Client/server or peer-to-peer?
  - Interact w/ performance, security, fault behavior.

Implementation

- How do clients/servers communicate?
  - Direct network communication is pretty painful

    - Want to hide network stuff from application logic
- Most systems organize distribution with some structuring framework(s)
    - RPC, RMI, DSM, MapReduce, etc.

Performance

- Distribution can hurt: network b/w and latency bottlenecks
    - Lots of tricks, e.g. caching, threaded servers
- Distribution can help: parallelism, pick server near client
    - Idea: scalable design
        - We would like performance to scale linearly with the addition of machines
        - `N x` servers `-> N x` total performance
- Need a way to divide the load by N
    - divide the state by N
        - split by user
        - split by file name
        - "sharding" or "partitioning"
- Rarely perfect `->` only scales so far
    - Global operations, e.g. search
    - Load imbalance
        - One very active user
        - One very popular file
            - `->` one server 100%, added servers mostly idle
            - `-> N x` servers `->` `1 x` performance

Fault tolerance

- Dropbox: ~10,000 servers; some fail
- Can I use my files if there's a failure?
    - Some part of network, some set of servers
- Maybe: replicate the data on multiple servers
    - Perhaps client sends every operation to both
    - Maybe only needs to wait for one reply
- *Opportunity:* operate from two "replicas" independently if partitioned?
- *Opportunity:* can 2 servers yield 2x availability **AND** 2x performance?

Consistency

- Contract w/ apps/users about meaning of operations
    - e.g. "read yields most recently written value"

    - hard due to partial failure, replication/caching, concurrency
- *Problem:* keep replicas identical
    - If one is down, it will miss operations
        - Must be brought up to date after reboot
    - If net is broken, *both* replicas maybe live, and see different ops
        - Delete file, still visible via other replica
        - *"split brain"* -- usually bad
- *Problem:* clients may see updates in different orders
    - Due to caching or replication
    - I make `grades.txt` unreadable, then TA writes grades to it
    - What if the operations run in different order on different replicas?
- Consistency often hurts performance (communication, blocking)
    - Many systems cut corners -- "relaxed consistency"
    - Shifts burden to applications

Labs

Focus: fault tolerance and consistency -- central to distributed systems.

- lab 1: MapReduce
- labs 2/3/4: storage servers
    - progressively more sophisticated (tolerate more kinds of faults)
        - progressively harder too!
    - patterned after real systems, e.g. MongoDB
    - Lab 4 has core of a real-world design for 1000s of servers

What you'll learn from the labs:

- easy to listen to lecture / read paper and think you understand
- building forces you to really understand
    - *"I hear and I forget, I see and I remember, I do and I understand"* (Confucius?)
- you'll have to do some design yourself
    - we supply skeleton, requirements, and tests
    - but we leave you substantial scope to solve problems your own way
- you'll get experience debugging distributed systems

Test cases simulate failure scenarios:

- distributed systems are tricky to debug: concurrency and failures
    - many client and servers operating in parallel
    - test cases make servers fail at the "most" inopportune time
- *think first* before starting to code!

    - otherwise your solution will be a mess
    - and/or, it will take you a lot of time
- code review
    - learn from others
    - judge other solutions

We've tried to ensure that the hard problems have to do w/ distributed systems:

- not e.g. fighting against language, libraries, etc.
- thus Go (type-safe, garbage collected, slick RPC library)
- thus fairly simple services (MapReduce, key/value store)

Lab 1: MapReduce

- help you get up to speed on Go and distributed programming
- first exposure to some fault tolerance
    - motivation for better fault tolerance in later labs
- motivating app for many papers
- popular distributed programming framework
- many descendants frameworks

Computational model

- aimed at document processing
    - split doc `-> K1 k, list<V1> values`
    - run `Map(K1 key, list<V1> values)` on each split `-> list<K2, V2> kvps`
    - run `Reduce(K2 key, list<V2> values)` on each partition `-> list<V2>`
    - merge result
- write a map function and reduce function
    - framework takes care of parallelism, distribution, and fault tolerance
- some computations are not targeted, such as:
    - anything that updates a document

Example: `wc`

- word count
- In Go's implementation, we have:
    - `func Map(value string) *list.List`
        - the input is *a split* of the file `wc` is called on
            - a split is just a partion of the file, as decided by MapReduce's splitter (can be customized, etc.)

    - returns a list of *key-value pairs*
      - the key is the word (like 'pen')
      - the value is 1 (to indicate 'pen' occurred once)
    - **Note:** there will be multiple `<'pen', 1>` entries in the list if 'pen' shows up more times
  - `func Reduce(key string, values *list.List) string`
    - the input is a key and a list of (all?) the values mapped to that key in the `Map()` phase
    - so here, we would expect a `Reduce('pen', [1,1,1,1])` call if pen appeared 4 times in the input file
      - **TODO**: not clear if it's also possible to get three reduce calls as follows:
        - `Reduce('pen', [1,1]) -> 2` + `Reduce('pen', [1,1]) -> 2`
        - `Reduce('pen', [2,2])`
        - the paper seems to indicate `Reduce` 's return value is just a list of values and so it seems that the association of those values with the key 'pen' in this case would be lost, which would prevent the 3rd `Reduce('pen')` call

Example: `grep`

- map phase
  - master splits input in `M` partitions
  - calls Map on each partition
    - `map(partition) -> list(k1,v1)`
    - search partition for word
    - produce a list with one item if word shows up, `nil` if not
    - partition results among `R` reducers
- reduce phase
  - Reduce job collects 1/R output from each Map job
  - all map jobs have completed!
  - `reduce(k1, v1) -> v2`
    - identity function: `v1` in, `v1` out
- merge phase
  - master merges `R` outputs

Performance

- number of jobs: `M x R` map jobs
- how much speed up do we get on `N` machines?
  - ideally: `N`

- bottlenecks:
  - stragglers
  - network calls to collect a Reduce partition
  - network calls to interact with FS
  - disk I/O calls

Fault tolerance model

- master is not fault tolerant
  - *assumption:* this single machine won't fail during running a MapReduce app
  - but many workers, so have to handle their failures
- assumption: workers are fail stop
  - they fail and stop (e.g., don't send garbled weird packets after a failure)
  - they may reboot

What kinds of faults might we want to tolerate?

- network:
  - lost packets
  - duplicated packets
  - temporary network failure
    - server disconnected
    - network partitioned
- server:
  - server crash+restart (master versus worker?)
  - server fails permanently (master versus worker?)
  - all servers fail simultaneously -- power/earthquake
  - bad case: crash mid-way through complex operation
    - what happens if we fail in the middle of map or reduce?
  - bugs -- but not in this course
    - what happens when bug in map or reduce?
    - same bug in Map over and over?
    - management software kills app
- malice -- but not in this course

Tools for dealing with faults?

- **retry** -- e.g. if packet is lost, or server crash+restart
  - packets (TCP) and MapReduce jobs
  - may execute MapReduce job twice: must account for this
- **replicate** -- e.g. if one server or part of net has failed
  - next labs

- **replace** -- for long-term health
  - e.g., worker

Retry jobs

- network falure: oops execute job twice
  - ok for MapReduce, because `map()/reduce()` produces same output
    - `map()/reduce()` are "functional" or "deterministic"
    - how about intermediate files?
      - atomic rename
- worker failure: may have executed job or not
  - so, we may execute job more than once!
  - but ok for MapReduce as long as `map()` and `reduce()` functions are deterministic
  - what would make `map() or reduce()` not deterministic?
  - is executing a request twice in general ok?
    - no. in fact, often not.
    - unhappy customer if you execute one credit card transaction several times
- adding servers
  - easy in MapReduce -- just tell master
  - hard in general
    - server may have lost state (need to get new state)
    - server may have rebooted quickly
      - may need to recognize that to bring server up to date
      - server may have a new role after reboot (e.g., not the primary)
    - these harder issues you would have to deal with to make the MapReduce master fault tolerant
    - topic of later labs

Lab 1 code

The lab 1 app (see `main/wc.go`):

- stubs for `map() and reduce()`
- you fill them out to implement word count (wc)
- how would you write grep?

The lab 1 sequential implementation (see `mapreduce/mapreduce.go`):

- demo: `run wc.go`
- code walk through start with `RunSingle()`

The lab 1 worker (see `mapreduce/worker.go`):

- the remote procedure calls (RPCs) arguments and replies (see `mapreduce/common.go`).
- Server side of RPC
    - RPC handlers have a particular signature
        - `DoJob`
        - `Shutdown`
- `RunWorker`
    - `rpcs.Register` : register named handlers -- so Call() can find them
    - `Listen` : create socket on which to listen for RPC requests
        - for distributed implementation, replace "unix" w. "tcp"
        - replace "me" with a `<dns,port>` tuple name
    - `ServeConn` : runs in a separate thread (why?)
        - serve RPC concurrently
        - a RPC may block
- Client side of RPC
    - `Register()`
- `call()` (see `common.go`)
    - make an RPC
    - lab code dials for each request
        - typical code uses a network connection for several requests
            - but, real must be prepared to redial anyway
            - a network connection failure, doesn't imply a server failure!
            - we also do this to introduce failure scenarios easily
            - intermittent network failures
            - just loosing the reply, but not the request

The lab 1 master (see mapreduce/master.go)

- You write it
- You will have to deal with distributing jobs
- You will have to deal with worker failures

6.824 2015 Lecture 2: Infrastructure: RPC and threads

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Remote Procedure Call (RPC)

- A key piece of distrib sys machinery; all the labs use RPC
- *Goal:* easy-to-program network communication
  - hides most details of client/server communication
  - client call is much like ordinary procedure call
  - server handlers are much like ordinary procedures
- RPC is widely used!

RPC ideally makes net communication look just like an ordinary *function call*:

```
Client
------
  z = fn(x, y)

Server
------
  fn(x, y) {
    compute
    return z
  }
```

RPC aims for this level of transparency

RPC message diagram

```
Client                          Server
------                          ------

        "fn", x, y
request ---------->
                            compute fn(x, y)

          z = fn(x, y)
         <------------- response
```

Software structure

```
  Client             Server
  ------             ------

 client app         handlers
   stubs           dispatcher
  RPC lib           RPC lib
    net <-----------> net
```

Stubs are sort of the fake client-side functions that look like the real `f(x, y)` but they just take care of packaging the arguments, sending them over the network and ask the server to compute `f(x, y)`. The stub can then receive the result over the network and return the value to the client code.

Examples from lab 1:

- `DoJob`
- `Register`

A few details of RPC

- *Marshalling:* format data into packets
    - Tricky for arrays, pointers, objects, etc.
    - Go's RPC library is pretty powerful!
    - some things you cannot pass/marshall: e.g., channels, functions
- *Binding:* how does client know who to talk to?
    - Might be a name service -- e.g. DNS
- *Threads:*
    - Client often has many threads, so `> 1` call outstanding, match up replies to calls
    - Handlers may be slow, so server often runs each in a thread

RPC problem: what to do about failures?

- e.g. lost packets, broken network, crashed servers, slow servers

What does a failure look like to the client's RPC library?

- It never sees a response from the server
    - Maybe packet was lost
- It does *not* know if the server saw the request!
    - Maybe server/net failed just before sending reply

Simplest scheme: *"at least once"* behavior

```
while true
    send req
    wait up to 10 seconds for reply
    if reply arrives
        return reply
    else
        continue
```

- RPC client library waits for response for a while
- If none arrives, re-send the request
- Do this a few times
- Still no response -- return an error to the application

Q: is "at least once" easy for applications to cope with?

Simple problem w/ at least once:

- Occurs with requests that are **not** *side-effect free*
- Client sends *"deduct $10 from bank account"* twice because it did not hear back for the first one

More subtle problem: what can go wrong with this client program?

- `Put("k", "v")` overwrites the value at `k` with `v`
- `Put("key", "value1")` -- an RPC to set key's value in a DB server
- `Put("key", "value2")` -- client then does a 2nd Put to same key

Example:

```
Client                               Server
------                               ------

put k, 10
            ----\
                 \
put k, 20   ---------------------->  k <- 20
                   \
                     ------------->  k <- 10

get k       ---------------------->

                10
            <----------------------
```

Note: This situation where client sends a request, server does some work and replies, but the reply is lost occurs frequently and will come up a lot in labs.

Is at-least-once ever OK?

- Yes: if it's OK to repeat operations, e.g. read-only op
- Yes: if application has its own plan for detecting duplicates
    - which you will need for Lab 1

Better RPC behavior: *"at most once"*

- *Idea:* server RPC code detects duplicate requests
  - returns previous reply instead of re-running handler
- Client includes *unique ID (XID)* with each request
  - uses same XID for re-send
- Server checks if XID has been seen before

Example:

```
if seen[xid]:
  r = old[xid]
else
  r = handler()
  old[xid] = r
  seen[xid] = true
```

Some at-most-once complexities

- How to ensure XID is unique?
  - big random number?
  - combine unique client ID (ip address?) with sequence #?
- Server must eventually discard info about old RPCs
  - When is discard safe?
  - *Idea:*
    - unique client IDs
    - per-client RPC sequence numbers
    - client includes *"seen all replies `<= X`"* with every RPC much like TCP sequence #s and ACKs
    - or only allow client one outstanding RPC at a time s.t. arrival of `seq+1` allows server to discard all `<= seq`
    - or client agrees to keep retrying for `< 5` minutes server discards after 5+ minutes
- How to handle duplicate request while original is still executing?
  - Server doesn't know reply yet; don't want to run twice
  - *Idea:* "pending" flag per executing RPC; wait or ignore

What if an at-most-once server crashes?

- if at-most-once duplicate info in memory, server will forget
  - and accept duplicate requests
- maybe it should write the duplicate info to disk?
- maybe replica server should also replicate duplicate info?

What about *"exactly once"*?

- *at-most-once* semantics plus unbounded retries plus fault-tolerant service

Go RPC is "at-most-once"

- open TCP connection
- write request to TCP connection
- TCP may retransmit, but server's TCP will filter out duplicates
- no retry in Go code (i.e. will NOT create 2nd TCP connection)
- Go RPC code returns an error if it doesn't get a reply
  - perhaps after a timeout (from TCP)
  - perhaps server didn't see request
  - perhaps server processed request but server/net failed before reply came back

Go's at-most-once RPC isn't enough for Lab 1

- it only applies to a single RPC call
- if worker doesn't respond, the master re-sends to it to another worker
  - but original worker may have not failed, and is working on it too
- Go RPC can't detect this kind of duplicate
  - No problem in lab 1, which handles at application level
  - In lab 2 you will have to protect against these kinds of duplicates

Threads

- threads are a fundamental server structuring tool
- you'll use them a lot in the labs
- they can be tricky
- useful with RPC
- called goroutines in Go

Thread = "thread of control"

- threads allow one program to (logically) do many things at once
- the threads share memory
- each thread includes some per-thread state:
  - program counter, registers, stack

Threading challenges:

- sharing data between thread
  - what if two threads modify same variable at same time?
  - what if one thread reads data another thread is changing?
  - these problems are often called *races*
  - need to protect invariants on shared data (Go: *mutex*)
- *coordination* between threads (Go: *channels*)
  - e.g. wait for all Map threads to finish
- *deadlocks*
  - thread 1 is waiting for thread 2
  - thread 2 is waiting for thread 1
  - easy detectable (unlike races)
- lock granularity
  - goarse-grained `->` little concurrency/parallelism
  - fine-grained `->` lots of concurrency, but race and deadlocks
- let's look at a toy RPC package to illustrate these problems

Look at today's handout -- l-rpc.go

Get it here.

- it's a toy RPC system
- illustrates threads, mutexes, channels
- it's a toy
  - assumes connection already open
  - only supports an integer arg, integer reply
  - doesn't deal with errors

`struct ToyClient`

- client RPC state
- mutex per `ToyClient`
- connection to server (e.g. TCP socket)
- xid -- unique ID per call, to match reply to caller
- `pending[]` -- multiple threads may call, need to find them
  - channel on which caller is waiting

`Call()`

- application calls `reply := client.Call(procNum, arg)`
- `procNum` indicates what function to run on server
- `WriteRequest` knows the format of an RPC msg
  - basically just the arguments turned into bits in a packet

- **Q:** why the mutex in `Call()`? what does `mu.Lock()` do?
- **Q:** could we move `xid := tc.xid` outside the critical section?
    - after all, we are not changing anything
    - [See diagram below]
- **Q:** do we need to `WriteRequest` inside the critical section?
    - note: Go says you are responsible for preventing concurrent map ops
    - that's one reason the update to pending is locked

Diagram:

`Listener()`

- runs as a background thread
- what is `<-` doing?
- not quite right that it may need to wait on chan for caller

Back to `Call()` ...

Q: what if reply comes back very quickly?

- could `Listener()` see reply before `pending[xid]` entry exists?
- or before caller is waiting for channel?

Q: should we put `reply := <-done` inside the critical section?

- why is it OK outside? after all, two threads use it.

Q: why mutex per `ToyClient`, rather than single mutex per whole RPC pkg?

Server's `Dispatcher()`

- note that the Dispatcher echos the xid back to the client
    - so that `Listener` knows which Call to wake up
- **Q:** why run the handler in a separate thread?
- **Q:** is it a problem that the dispatcher can reply out of order?

`main()`

- note registering handler in `handlers[]`
- what will the program print?

When to use shared memory (and locks) vs when to use channels?

- here is my opinion
- use channels when you want one thread to explicitly wait for another
    - often wait for a result, or wait for the next request

    - e.g. when client `Call()` waits for `Listener()`
- use shared memory and locks when the threads are not intentionally
    - directly interacting, but just happen to r/w the same data
    - e.g. when `Call()` uses `tc.xid`

Go's "memory model" requires explicit synchronization to communicate!

This code is not correct:

```
var x int
done := false
go func() { x = f(...); done = true }
while done == false { }
```

It's very tempting to write, but the Go spec says it's undefined use a channel or `sync.WaitGroup` instead

Study the Go tutorials on *goroutines* and *channels*.

6.824 2015 Lecture 3: Primary/Backup Replication

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Today

- Replication
- Remus case study
- Lab 2 introduction

Fault tolerance

- We'd like a service that continues despite failures!
- **Definitions:**
  - *Available* -- still usable despite [some class of] failures
  - *Correct* -- act just like a single server to clients
    - A lot of issues that come up have to do with *correctness*
- Very hard!
- Very useful!

Need a failure model: what will we try to cope with?

- *Most common:* Independent fail-stop computer failures
  - *fail-stop failures:* computed correctly for a while and then stopped
    - as opposed to computing incorrectly (different situation)
  - have to assume independence of failures
    - (o.w. we could have primary fail `=>` backup fail `=>` fffffuu....)
  - Remus further assumes only one failure at a time
- *Another model:* Site-wide power failure (and eventual reboot)
- (Network partition)
- No bugs, no malice

Core idea: replication

- *Two* servers (or more)

- Each replica keeps state needed for the service
- If one replica fails, others can continue

Example: fault-tolerant MapReduce master

- Lab 1 workers are already fault-tolerant, but not master
- `[Diagram: M1, M2, workers]`
- State:
    - worker list
    - which jobs done
    - which workers idle
    - TCP connection state
    - program counter

Big questions

- What *state* to replicate?
    - *Example:* Remus replicates all of RAM and the CPU state
- How does replica get state?
- When to *cut over* to backup?
    - Is primary really down or is just the network down?
- Are anomalies visible at cut-over?
    - What will clients see?
- How to repair / re-integrate?
    - How to get a new backup?

Two main approaches:

1. **State transfer**
    - "Primary" replica executes the service
    - Primary sends [new] state to backups
    - *Example:* Remus
2. **Replicated state machine**
    - All replicas (primary and backup) execute all operations
    - If same start state & same operations & same order & deterministic & *then* `=>` same end state
    - *Ops* are transferred and not the state

State transfer is simpler

- But state may be large, slow to transfer

- *Remus* uses state transfer

Replicated state machine can be more efficient

- If operations are small compared to data
- But complex, e.g. order on multi-core, determinism
  - Hard to make sure everyone got to the same state
  - Determinism can be problematic (time, threads, etc.)
- Labs use replicated state machines

Remus: High Availability via Asynchronous Virtual Machine Replication, NSDI 2008

Very ambitious system

- Whole-system replication
- Completely *transparent* to applications and clients
- High availability for any existing software
- Would be magic if it worked well!
- *Failure model:*
  1. Independent hardware faults
  2. Site-wide power failure

Plan 1 (slow, broken):

- `[Diagram: app, O/S, Remus underneath]`
- two machines, *primary* and *backup*; plus net and other machines
- primary runs o/s and application s/w, talks to clients, etc.
- backup does *not* initially execute o/s, applications, etc.
  - it only executes some Remus code
- a few times per second:
  - pause primary
  - copy **entire RAM**, registers, disk to backup
    - 10Gbps = 1GB/s network bandwidth
    - 100MB/s disk bandwidth
    - network bandwidth limits RAM transfer rate
    - disk bandwidth limits disk transfer rate
  - resume primary
- if primary fails:
  - start backup executing!

Q: Is Plan 1 correct (as described above)?

- i.e. does it look just like a single reliable server?
- No:
  - client sends write req. to primary, primary replies before backup had a chance to copy the new state
  - primary fails, backup takes over, but it does not reflect the last write req.
  - client will be screwed because his write was lost

Q: What will outside world see if primary fails and replica takes over?

- Will backup have same state as last visible on primary?
- Might a client request be lost? Executed twice?
- Yes: see above question

Q: How to decide if primary has failed?

Q: How will clients know to talk to backup rather than primary?

Q: What if site-wide power failure?

- Primary is running some o/s, has a plan for reboot from disk "crash-consistent"

Q: What if primary fails while sending state to backup?

- i.e. backup is mid-way through absorbing new state?

Q: What if primary gets request, sends checkpoint to backup, and just before replying primary fails?

- TCP layer will take care of this? If client retransmits request, that could be problematic (side effects). So hopefully TCP kicks in and notices that no reply came back. How? Primary was just about to reply, but Remus held the reply in the buffer. Backup will have same state so it'll think it has replied and wait for an ACK from the client, which will never come because the client got nothing. Thus, backup will retransmit the packets that the primary never had a chance to and finally get the ACK from the client.

Q: Is Plan 1 efficient?

- Can we eliminate the fact that backup *state* trails the primary?
  - Seems very hard!
  - Primary would have to tell backup (and wait) on every instruction.
- Can we *conceal* the fact that backup's state lags primary?
  - Prevent outside world from *seeing* that backup is behind last primary state
    - e.g. prevent primary sent RPC reply but backup state doesn't reflect that RPC
    - e.g. MapReduce `Register()` RPC, which it would be bad for backup to forget

  - *Idea:* primary "holds" output until backup state catches up to output point
    - e.g. primary receives RPC request, processes it, creates reply packet, but Remus holds reply packet until backup has received corresponding state update

Remus epochs, checkpoints

1. Primary runs for a while in Epoch 1 (E1), holding E1's output
2. Primary pauses
3. Primary copies RAM+disk changes from E1 to local buffer
4. Primary resumes execution in E2, holding E2's output
5. Primary sends checkpoint of RAM+disk to backup
6. Backup copies all to separate RAM, then applies, then ACKs
7. Primary releases E1's output
8. Backup applies E1's changes to RAM and disk

If primary fails, backup finishes applying last epoch's disk+RAM, then starts executing

Q: Any externally visible anomalies?

Q: What if primary receives + executes a request, crashes before checkpoint? backup won't have seen request!

- That's fine as long as primary did not reply to that request: client will just send request again

Q: If primary sends a packet, then crashes, is backup guaranteed to have state changes implied by that packet?

- Yes. That's the whole point of keeping the sent network packets buffered until the backup is up to date.

Q: What if primary crashes partway through release of output? must backup re-send? How does it know what to re-send?

Q: How does Remus decide it should switch to backup?

- Naive mechanism: If the primary stops talking to the backup, then something went wrong.

Q: Are there situations in which Remus will incorrectly activate the backup? i.e. primary is actually alive

- Network partition...

Q: When primary recovers, how does Remus restore replication? Needed, since eventually active ex-backup will itself fail

Q: What if *both* fail, e.g. site-wide power failure?

- RAM content will be lost, but disks will probably survive
- After power is restored, reboot guest from one of the disks
  - O/S and application recovery code will execute
- disk must be "crash-consistent"
  - So probably not the backup disk if was in middle of installing checkpoint
- disk shouldn't reflect any held outputs (... why not?)
  - So probably not the primary's disk if was executing
- I do not understand this part of the paper (Section 2.5)
  - Seems to be a window during which neither disk could be used if power failed
    - primary writes its disk during epoch
    - meanwhile backup applies last epoch's writes to its disk

Q: In what situations will Remus likely have good performance?

Q: In what situations will Remus likely have low performance?

Q: Should epochs be short or long?

Remus evaluation

- *Summary:* 1/2 to 1/4 native speed
- Checkpoints are big and take time to send
- Output hold limits speed at which clients can interact

Why so slow?

- Checkpoints are big and take time to generate and send
  - 100ms for SPECweb2005 -- because many pages written
- So inter-checkpoint intervals must be long
- So output must be held for quite a while
- So client interactions are slow
  - Only 10 RPCs per second per client

How could one get better performance for replication?

- Big savings possible with application-specific schemes:
  - just send state really needed by application, not all state

  - send state in optimized format, not whole pages
  - send operations if they are smaller than state
- likely *not* transparent to applications
  - and probably not to clients either

Primary-backup replication in Lab 2

Outline

- simple key/value database
- primary and backup
- *replicated state machine:* replicate by primary sending each operation to backups
- tolerate network problems, including partition
  - either keep going, correctly
  - or suspend operations until network is repaired
- allow replacement of failed servers
- you implement essentially all of this (unlike lab 1)

"View server" decides who primary `p` and backup `b` are

- *Main goal:* avoid "split brain" -- disagreement about who primary is
- Clients and servers ask view server
- They don't make independent decisions

Repair

- view server can co-opt "idle" server as `b` after old `b` becomes `p`
- primary initializes new backup's state

Key points:

1. Only one primary at a time!
2. The primary must have the latest state!

We will work out some rules to ensure these

View server

- Maintains a sequence of "views"

Example:

```
view #, primary, backup
0:      --       --
1:      S1       --
2:      S1       S2
4:      S2       --
3:      S2       S3
```

- Monitors server liveness
  - each server periodically sends a ping RPC (more like a heartbeat)
  - *"dead"* if missed `N` pings in a row
  - *"live"* after single ping
- Can be more than two servers pinging view server
  - if more than two, *"idle"* servers
- If primary is dead:
  - new view with previous backup as primary
- If backup is dead, or no backup
  - new view with previously idle server as backup
- OK to have a view with just a primary, and no backup
  - But -- if an idle server is available, make it the backup

How to ensure new primary has up-to-date replica of state?

- Only promote previous backup
  - i.e. don't make an idle server the primary
- Backup must remember if it has been initialized by primary
  - If not, don't function as primary even if promoted!

Q: Can more than one server think it is primary?

```
1: S1, S2
   net broken, so viewserver thinks S1 dead but it's alive
2: S2, --
now S1 alive and not aware of view #2, so S1 still thinks it is primary
AND S2 alive and thinks it is primary
=> split brain, no good
```

How to ensure only one server acts as primary?

...even though more than one may *think* it is primary.

"Acts as" `==` executes and responds to client requests

The basic idea:

```
1: S1 S2
2: S2 --
S1 still thinks it is primary
S1 must forward ops to S2
S2 thinks S2 is primary
so S2 must reject S1's forwarded ops
```

The rules:

1. Primary in view `i` must have been primary or backup in view `i-1`
2. Primary must wait for backup to accept each request
   - **Q:** What if there's no backup or the backup doesn't know it's a backup?
   - **A:** Primary can't make progress without a backup if it's part of the view, so it just waits
   - **A:** If the view is updated and the backup is taken out of the view then primary can operate in "dangerous mode" without a backup
3. Non-backup must reject forwarded requests
4. Non-primary must reject direct client requests
5. Every operation must be before or after state transfer

Example:

```
1: S1, S2
   viewserver stops hearing Pings from S1
2: S2, --
   it may be a while before S2 hears about view #2

before S2 hears about view #2
  S1 can process ops from clients, S2 will accept forwarded requests
  S2 will reject ops from clients who have heard about view #2
after S2 hears about view #2
  if S1 receives client request, it will forward, S2 will reject
    so S1 can no longer act as primary
  S1 will send error to client, client will ask viewserver for new view,
     client will re-send to S2
the true moment of switch-over occurs when S2 hears about view #2
```

How can new backup get state?

- e.g. all the keys and values
- if S2 is backup in view `i`, but was not in view `i-1`,
  - S2 should ask primary to transfer the complete state

Rule for state transfer:

- every operation (`Put/Get/Append`) must be either before or after state xfer
  - `==` state xfer must be atomic w.r.t. operations
- either
  - op is before, and xferred state reflects op

  - op is after, xferred state doesn't reflect op, primary forwards op after state

Q: Does primary need to forward `Get()` 's to backup?

- After all, `Get()` doesn't change anything, so why does backup need to know?
- and the extra RPC costs time
- has to do with ensuring there's just one primary:
  - suppose there's two primaries by accident (P and P' both think they are primaries)
    - how can this occur? network partition?
  - suppose client sends a Get request to the wrong primary P'
  - then P' will try to fwd the request to P (which P' thinks it's the backup)
  - then P will tell P': *"Hey, bug off, I'm the primary"*

Q: How could we make primary-only `Get()` 's work?

Q: Are there cases when the Lab 2 protocol cannot make forward progress?

- View service fails
- Primary fails before backup gets state
- We will start fixing those in Lab 3

6.824 2015 Lecture 4: "Flat Datacenter Storage" Case Study

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Flat datacenter storage

Flat Datacenter Storage, *Nightingale, Elson, Fan, Hofmann, Howell, Suzue*, OSDI 2012

Why are we looking at this paper?

- Lab 2 wants to be like this when it grows up
  - though details are all different
- fantastic performance -- world record cluster sort
- good systems paper -- details from apps all the way to network

What is FDS?

- a cluster storage system
- stores giant blobs -- 128-bit ID, multi-megabyte content
- clients and servers connected by network with high bisection bandwidth
- for big-data processing (like MapReduce)
  - cluster of 1000s of computers processing data in parallel

High-level design -- a common pattern

- lots of clients
- lots of storage servers ("tractservers")
- lots of bandwidth between any two servers
- data is stored in blobs
  - addressed by 128bit IDs
  - further split into tracts
    - numbered from 0
    - 8MB sized
- partition the data
- master ("metadata server") controls partitioning
- replica groups for reliability

- tract table locator (TLT) stores a bunch entries
    - in a `k` -replicated system, each entry has `k` tractservers
- how to find where a tract `t` for blob `b` is?
    - compute TLT entry as `(h(b) + t) mod len(tlt)`
        - and you'll get a list of servers in that entry
    - blob metadata is *distributed* and NOT stored in the TLT
- how to write a tract from a blob?
    - look it up as described above
    - send write to all servers in TLT entry
    - only acknowledge write to client if *all* servers replied
- how to read a tract from a blob?
    - look it up as described above
    - send read to *a random* server in the TLT entry

Why is this high-level design useful?

- 1000s of disks of space
    - store giant blobs, or many big blobs
- 1000s of servers/disks/arms of parallel throughput
- can expand over time -- reconfiguration
- large pool of storage servers for instant replacement after failure

Motivating app: MapReduce-style sort

- a mapper reads its split `1/M'th` of the input file (e.g., a tract)
    - map emits a `<key, record>` for each record in split
    - map partitions keys among `R` intermediate files (`M*R` intermediate files in total)
- a reducer reads 1 of `R` intermediate files produced by each mapper
    - reads `M` intermediate files (of `1/R` size)
    - sorts its input
    - produces `1/R'th` of the final sorted output file (`R` blobs)
- FDS sort
    - FDS sort does not store the intermediate files in FDS
    - a client is both a mapper and reducer
    - FDS sort is not locality-aware
        - in mapreduce, master schedules workers on machine that are close to the data
        - e.g., in same cluster later versions of FDS sort uses more fine-grained work assignment e.g., mapper doesn't get 1/N of the input file but something smaller deals better with stragglers

The abstract's main claims are about performance.

- They set the world-record for disk-to-disk sorting in 2012 for MinuteSort
  - 1,033 disks and 256 computers (136 tract servers, 120 clients)
  - 1,401 Gbyte in 59.4s

Q: Does the abstract's 2 GByte/sec per client seem impressive?

- how fast can you read a file from Athena AFS? (abt 10 MB/sec)
- how fast can you read a typical hard drive?
- how fast can typical networks move data?

Q: Abstract claims recover from lost disk (92 GB) in 6.2 seconds

- that's 15 GByte / sec
- impressive?
- how is that even possible? that's 30x the speed of a disk!
- who might care about this metric?

What should we want to know from the paper?

- API?
- layout?
- finding data?
- add a server?
- replication?
- failure handling?
- failure model?
- consistent reads/writes? (i.e. does a read see latest write?)
  - Not in FDS: "The current protocol for replication depends upon the client to issue all writes to all replicas. This decision means that FDS provides weak consistency guarantees to clients. For example, if a client writes a tract to 1 of 3 replicas and then crashes, other clients reading different replicas of that tract will observe differing state."
  - "Writes are not guaranteed to be committed in order of issue. Applications with ordering requirements are responsible for issuing operations after previous acknowledgments have been received, rather than concurrently. FDS guarantees atomicity: a write is either committed or failed completely"
- config mgr failure handling?
- good performance?
- useful for apps?

API

- Figure 1
- 128-bit blob IDs
- blobs have a length
- only whole-tract read and write -- 8 MB

Q: Why are 128-bit blob IDs a nice interface?

- Why not file names?

Q: Why do 8 MB tracts make sense?

- (Figure 3...)

Q: What kinds of client applications is the API aimed at?

- and not aimed at?

Layout: how do they spread data over the servers?

- Section 2.2
- break each blob into 8 MB tracts
- TLT maintained by metadata server
  - has `n` entries
  - for blob `b` and tract `t`, `i = (hash(b) + t) mod n`
  - `TLT[i]` contains list of tractservers w/ copy of the tract
- clients and servers all have copies of the latest TLT table

Example four-entry TLT with no replication:

```
0: S1
1: S2
2: S3
3: S4
suppose hash(27) = 2
then the tracts of blob 27 are laid out:
S1: 2 6
S2: 3 7
S3: 0 4 8
S4: 1 5 ...
FDS is "striping" blobs over servers at tract granularity
```

Q: hy have tracts at all? Why not store each blob on just one server?

- What kinds of apps will benefit from striping?
- What kinds of apps won't?

Q: How fast will a client be able to read a single tract?

Q: Where does the abstract's single-client 2 GB number come from?

Q: Why not the UNIX i-node approach?

- store an array per blob, indexed by tract #, yielding tractserver
- so you could make per-tract placement decisions
  - e.g. write new tract to most lightly loaded server

Q: Why not `hash(b + t)` ?

Q: How many TLT entries should there be?

- how about `n = number of tractservers` ?
- why do they claim this works badly? Section 2.2

The system needs to choose server pairs (or triplets &c) to put in TLT entries

- For replication
- Section 3.3

Q: How about:

```
0: S1 S2
1: S2 S1
2: S3 S4
3: S4 S3
...
```

- Why is this a bad idea?
- How long will repair take?
- What are the risks if two servers fail?

Q: why is the paper's `n^2` scheme better?

Example:

```
0: S1 S2
1: S1 S3
2: S1 S4
3: S2 S1
4: S2 S3
5: S2 S4
...
```

- TLT with `n^2` entries, with every server pair occuring once
- How long will repair take?
- What are the risks if two servers fail?

Q: Why do they actually use a minimum replication level of 3?

- Same `n^2` table as before, third server is randomly chosen
- What effect on repair time?

- What effect on two servers failing?
- What if three disks fail?

Adding a tractserver

- To increase the amount of disk space / parallel throughput
- Metadata server picks some random TLT entries
- Substitutes new server for an existing server in those TLT entries

Extending a tract's size

- Newly created blobs have a length of 0 tracts
- Applications must extend a blob before writing past the end of it.
- The extend operation is atomic, is safe to execute concurrently with other clients, and returns the new size of the blob as a result of the client's call.
- A separate API tells the client the blob's current size.
- Extend operations for a blob are sent to the tractserver that owns that blob's metadata tract.
- The tractserver serializes it, atomically updates the metadata, and returns the new size to each caller.
- If all writers follow this pattern, the extend operation provides a range of tracts the caller may write without risk of conflict. Therefore, the extend API is functionally equivalent to the Google File System's "atomic append."
- Space is allocated lazily on tractservers, so tracts claimed but not used do not waste storage.

How do they maintain `n^2` plus one arrangement as servers leave join?

Unclear.

Q: How long will adding a tractserver take?

Q: What about client `write`'s while tracts are being transferred?

- receiving tractserver may have copies from client(s) and from old srvr
- how does it know which is newest?

Q: What if a client reads/writes but has an old tract table?

- tractservers tell him

Replication

- A writing client sends a copy to each tractserver in the TLT.
- A reading client asks one tractserver.

Q: Why don't they send writes through a primary?

- puts a lot of work on a primary? has to lookup and know TLT
- goal is not to have just one backup for a primary, it's to replicate and strip data effectively across many disks

Q: What problems are they likely to have because of lack of primary?

- Why weren't these problems show-stoppers?

What happens after a tractserver fails?

- Metadata server stops getting heartbeat RPCs
- Picks random replacement for each TLT entry failed server was in
- New TLT gets a new version number
- Replacement servers fetch copies

Example of the tracts each server holds:

```
S1: 0 4 8 ...
S2: 0 1 ...
S3: 4 3 ...
S4: 8 2 ...
```

Q: Why not just pick one replacement server?

- it will have to take in a lot of writes for the lost data `=>` bad perf.

Q: How long will it take to copy all the tracts?

Q: If a tractserver's net breaks and is then repaired, might srvr serve old data?

Q: If a server crashes and reboots with disk intact, can contents be used?

- e.g. if it only missed a few writes?
- 3.2.1's "partial failure recovery"
- but won't it have already been replaced?
- how to know what writes it missed?

Q: When is it better to use 3.2.1's partial failure recovery?

What happens when the metadata server crashes?

Q: While metadata server is down, can the system proceed?

- yes, clients who have the TLT can go on

Q: Is there a backup metadata server?

- not in the paper, they said they might use Paxos for replication
- **TODO:** not clear why replicating the metadata server would lead to consistency problems

Q: How does rebooted metadata server get a copy of the TLT?

- Eh, maybe it has it on disk?
- Maybe it just simply reconstructs it from all the heartbeats?

Q: Does their scheme seem correct?

- how does the metadata server know it has heard from all tractservers?
    - it doesn't, it just adds servers as they send heartbeats
- how does it know all tractservers were up to date?
    - **TODO:** Up to date with what?

Random issues

Q: Is the metadata server likely to be a bottleneck?

- hard to tell. what's the use case?
- if you have a client w/ that has memory to remember TLT then he only contacts metadata server once and then starts doing all of his reads/writes
- if you have a lot of clients joining the system, or coming back but forgetting the TLT (because of lack of storage maybe), then the metadata server would be in use heavily
    - however, this won't affect the bandwidth the clients get once they downloaded the TLT

Q: Why do they need the scrubber application mentioned in 2.3?

- why don't they delete the tracts when the blob is deleted?
    - faster to do GC, rather than scheduling & executing deletes?
- can a blob be written after it is deleted?
    - **TODO:** not sure, seems like yes, because the metadata for that blob is in tract -1 and I don't think `WriteTract` checks the metadata before every write, so you could maybe have races

Performance

Q: How do we know we're seeing "good" performance? What's the best you can expect?

- best you can expect is to take each disks bandwidth and have the system's bandwidth be `# of disk * disk bandwidth`

Q: Limiting resource for 2 GBps single-client?

- assuming this is end of 5.2
- 30 tractservers means maximum of 30 * 130MB/s = 3.9GBps
- so limiting resource is network bandwidth

Q: Figure 4a: Why starts low? Why goes up? Why levels off? Why does it level off at that particular performance?

- starts low because single client bandwidth is limited
- goes up b.c. as # of clients is increased each one adds more bandwidth to the system
- levels off because at some point the client bandwidth > server's bandwidth
- why levels off at 32 GBps for `x` clients w/ 516 disks?
    - Figure 3 suggests a 10,000 RPM disk can read 5MB chunks at around 130MB/s
        - writes are similar
    - not clear from logarithmic scale graph what `x` is
        - `10 < x < 50` (maybe `25 < x < 50`?)
    - `516 disks * 130MB/s = 67 GBps`, so seems like best case performance should've leveled off at more than 32 GBps?
        - in reality not all disks are 130MB/s maybe? (only the 10,000rpm SAS onese were that fast)
        - in reality multiple disks on a single node might make that number smaller maybe?
        - anyway, something like `x=40` clients would have `40 * 10Gbps = 40 * 1.25GBps = 50 Gbps` which is higher than the actual (claimed) bandwidth of the server of 32 GBps

Q: Figure 4b shows random r/w as fast as sequential (Figure 4a). Is this what you'd expect?

- Yes. Random R/W requests for different tracts go to different servers, just like sequential ones do `=`> no difference

Q: Why are writes slower than reads with replication in Figure 4c?

- A write is sent to all tract servers? Not over until all of them reply.
    - w/ higher number of clients writing `=>` more work done by each server
    - Paper says: "As expected, the write bandwidth is about onethird of the read bandwidth since clients must send three copies of each write"
- A read is sent to just one?

Q: Where does the 92 GB in 6.2 seconds come from?

- Table 1, 4th column
- that's 15 GB / second, both read and written
- 1000 disks, triple replicated, 128 servers?
- what's the limiting resource? disk? cpu? net?

How big is each sort bucket?

- i.e. is the sort of each bucket in-memory?
- 1400 GB total
- 128 compute servers
- between 12 and 96 GB of RAM each
- hmm, say 50 on average, so total RAM may be 6400 GB
- thus sort of each bucket is in memory, does not write passes to FDS
- thus total time is just four transfers of 1400 GB
    - client limit: `128 * 2 GB/s = 256 GB / sec`
    - disk limit: `1000 * 50 MB/s = 50 GB / sec`
- thus bottleneck is likely to be disk throughput

6.824 2015 Lecture 5: Paxos

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Intro

Starting a new group of lectures on stronger fault tolerance

- Today:
  - cleaner approach to replication: RSM via Paxos
  - Lab 3
- Subsequent lectures:
  - How to use Paxos to build systems (Harp, EPaxos, Spanner)

Paxos

Links:

- Paxos Made Simple, by Leslie Lamport, 2001
- Simple explanations from Quora
- Neat Algorithms - Paxos
- Paxos Replicated State Machines as the Basis of a High-Performance Data Store
- Paxos notes
- Paxos made simple paper review
- On some subtleties of Paxos

Recall: RSM

- maintain replicas by executing operations in the same order
- requires all replicas to agree on the (set and) order of operations

Lab 2 critique

- primary/backup with viewserver
- **pro:**
  - conceptually simple
  - just two msgs per op (request, reply)
  - primary can do computation, send result to backup
  - only two k/v servers needed to tolerate one failure

- works with network partition
- **con:**
    - ViewServer is a *single point of failure*
    - order can be messy, e.g. new view, data to backup, ack, &c
    - tension if backup is slow / temporarily unavail
        1. primary can wait for backup -- slow
        2. viewserver can declare backup dead -- expensive, hurts fault tolerance

We would like a general-purpose ordering scheme with:

- no single point of failure
- graceful handling of slow / intermittent replicas
- handling of network partitions

Paxos will be a key building block for this.

- some number of nodes participate in *an instance of Paxos*
    - **Q:** What is this *instance*?
    - **A:** *"Each new command requires a separate Paxos agreement, which the paper calls an instance. So the database replicas might agree that the first command to execute is 'command one', and they use Paxos to agree on that. Then that instance of Paxos is done. A while later another client sends 'command two'; the replicas start up an entirely separate instance of Paxos to agree on this second client command.'* --RTM
- each node knows the address of every other node for that instance
- each instance of Paxos can reach consensus on at most one value typically, a system uses many instances of Paxos each instance usually decides one operation assumptions: asynchronous, non-Byzantine

What does Paxos provide? How does it work?

- **"black-box"** interface to a Paxos instance, on each node:
    - Propose a value (e.g., operation)
    - Check what value has been decided, if any
    - [Lab 3A: `src/paxos/paxos.go` : Start, Status]
- **Correctness:**
    - if agreement reached, all agreeing nodes see same value
- **Fault-tolerance:**
    - can tolerate non-reachability of a minority of nodes (correctness implies they won't agree at all)
- **Liveness:**
    - a majority must be alive and able to communicate reliably (minorities are not live)

How to build a system using Paxos?

1. Primary/Backup like Lab 2, but use Paxos to replicate the ViewServer
   - [next Tuesday's lecture will be about such a system]
2. *Lab 3*: no ViewServer, all replicas use Paxos instead of primary/backup

Replicating either the ViewServer or K/V server with Paxos is similar.

Will look at a sketch of how to do a Paxos-based K/V server.

The **basic idea**:

- [Diagram: clients, replicas, log in each replica, K/V layer, Paxos layer]
- no viewserver
- three replicas
- clients can send RPCs to any replica (not just primary)
- server appends each client op to a replicated *log* of operations
  - `Put`, `Get` (and more later)
- log entries (instances) are numbered sequentially
- Paxos ensures agreement on content of each log entry
- separate Paxos agreement for each of these log entries
  - separate *instance* of Paxos algorithm is run for log entry # `i`
  - **Q:** Can one log entry be agreed on at the same time with another? What if they depend on one another like `Put(k1, a)` and `Append(k1, b)`?
  - **A:** Yes! They can be agreed upon on the same time.
  - **A:** you can have agreed on log entry # `i` before agreeing on log entry # `i+1`
    - This means the reply associated with the `Get` or `Put` request in log entry `i+1` will have to wait for the other log entries to be set (interesting)
- servers can throw away log entries that all other servers have agreed on (and responded to?)
  - but if a server crashes, the other servers will know to keep their log entries around for when it comes back
- protocol does **not** require designated proposers or leaders for correctness
  - these only help w/ performance
  - low probability of proposing "livelock" that can be overcome by having proposers wait a random amount of time
- once a Paxos node agrees on a value it never changes its mind

Example:

- client sends `Put(a, b)` to `S1`
- `S1` picks a log entry 3
- `S1` uses Paxos to get all servers to agree that entry 3 holds `Put(a,b)`

Example:

- client sends `Get(a)` to `S2`
- `S2` picks log entry 4
- `S2` uses Paxos to get all servers to agree that entry 4 holds `Get(a)`
- `S2` scans log up to entry 4 to find latest `Put(a, ...)`
    - **TODO:** `O(n)` worst case for doing a `Get` because you can have `Put` followed by a gazillion `PutAppend` 's (or you can have just one `Put` stored way back?).
        - Can the replicas index their log? I suppose. If they all store it fully.
- `S2` replies with that value
    - `S2` can cache content of DB up through last log scan

Q: Why a log?

- Why not require all replicas to agree on each op in lock-step?
- Allows one replica to fall behind, then catch up
    - e.g. if it is slow
    - other replicas do not have to wait
- Allows one replica to crash and catch up
    - if it keeps state on disk
    - can replay missed operations
- Allows pipelining/overlap of agreement
    - agreement turns out to require multiple message rounds

Q: What about agreement -- we need all replicas to have same op in each log slot

- Provided by Paxos, as we will see next

Agreement is hard (1):

- May be multiple proposals for the op in a particular log slot
- `Sx` (server `x`) may initially hear of one, `Sy` may hear of another
- Clearly one must later change its mind
- Thus: multiple rounds, tentative initially
- How do we know when agreement is permanent -- no longer tentative?

Agreement is hard (2):

- **TODO:** If `S1` and `S2` agree, and `S3` and `S4` don't respond, are we done?
- Agreement has to be able to complete even w/ failed servers
- We can't distinguish failed server from network partition
- So maybe `S3` / `S4` are partitioned have "agreed" on a different operation!

Two **main ideas** in Paxos:

1. Many rounds may be required but they will converge on one value
2. A majority is required for agreement -- prevent "split brain"
    - *Key point*: any two majorities overlap
    - so any later majority will share at least one server w/ any earlier majority
    - so any later majority can find out what earlier majority decided
        - **TODO:** How?

Lab 3B K/V server creates a separate Paxos instance for each client `Put` , `Get`

- rest of lecture focuses on agreement for a specific instance

Paxos sketch

- each node consists of three logical entities:
    - **proposer**
    - **acceptor**
    - **learner**
- each proposer wants to get agreement on its value
    - could try to use a "designated leader" to avoid dueling proposers
    - OK to have multiple proposers, so leader election can be approximate
- proposer contacts acceptors, tries to assemble a majority
    - if a majority respond, we're done
- in our K/V server example, roughly:
    - proposer gets RPC from client, proposes operation
    - acceptors are internal to Paxos, help decide consensus
    - learner figures out what operation was decided to run, responds to client

Broken strawman: can we do Paxos in a single round?

- acceptor "accepts" the first value that it hears from proposer
- when is consensus reached?
    - can we take the value with the most votes?
    - no, need a majority of accepts for the same value: `floor(n/2)+1`
    - otherwise, consensus on 2 different values (lossy/partitioned network)
- *Problem:*
    - suppose we have 3 servers: `S1` , `S2` , `S3`
    - what if each server proposes + accepts its own value?
        - no majority, stuck
        - but maybe we can detect this situation and recover?
    - *Worse:* `S3` crashes `->` we may have reached majority, but we'll never know
- need a way for acceptors to change their mind, if no consensus reached yet

Basic Paxos exchange

```
    proposer          acceptors

          prepare(n) ->
       <- prepare_ok(n, n_a, v_a)

          accept(n, v') ->
       <- accept_ok(n)

          decided(v') ->
```

Why `n` ?

- to distinguish among multiple rounds, e.g. proposer crashes, simul props
- want later rounds to supersede earlier ones
- numbers allow us to compare early/late
- `n` values must be unique and roughly follow time
- `n = <time, server ID>`
  - e.g., ID can be server's IP address
- *"round"* is the same as *"proposal"* but completely different from "instance"
  - round/proposal numbers are *WITHIN* a particular instance

Definition: server S *accepts* `n/v`

- it responded `accept_ok` to `accept(n, v)`

Definition: `n/v` is *chosen*

- a majority of servers accepted `n/v`

The **crucial property:**

- if a value was chosen, any subsequent choice must be the same value
  - i.e. protocol must not change its mind
  - maybe a different proposer &c, but same value!
  - this allows us to freely start new rounds after crashes &c
- tricky b/c *"chosen"* is system-wide property
  - e.g. majority accepts, then proposer crashes
    - **TODO:** What happens here?
  - *no node can tell locally that agreement was reached*

So:

- proposer doesn't send out value with `prepare`
  - **TODO:** How is any value accepted by an acceptor then?
- acceptors send back any value they have already accepted

- if there is one, proposer proposes that value
  - to avoid changing an existing choice
- if no value already accepted,
  - proposer can propose any value (e.g. a client request)
- proposer must get `prepare_ok` from majority
  - to guarantee intersection with any previous majority,
  - to guarantee proposer hears of any previously chosen value

Now the protocol -- see the handout

```
proposer(v):
  choose n, unique and higher than any n seen so far
  send prepare(n) to all servers including self
  if prepare_ok(n, n_a, v_a) from majority:
    v' = v_a with highest n_a; choose own v otherwise
    send accept(n, v') to all
    if accept_ok(n) from majority:
      send decided(n, v') to all

acceptor state:
  must persist across reboots
  n_p (highest prepare seen)
  n_a, v_a (highest accept seen)

acceptor's prepare(n) handler:
  if n > n_p
    n_p = n
    reply prepare_ok(n, n_a, v_a)
  else
    reply prepare_reject

acceptor's accept(n, v) handler:
  if n >= n_p
    n_p = n
    n_a = n
    v_a = v
    reply accept_ok(n)
  else
    reply accept_reject
```

Example 1 (normal operation):

```
`S1`, `S2`, `S3` but `S3` is dead or slow

`S1`: -> starts proposal w/ n=1 v=A
`S1`: <- p1   <- a1vA    <- dA
`S2`: <- p1   <- a1vA    <- dA
`S3`: dead...

"p1" means Sx receives prepare(n=1)
"a1vA" means Sx receives accept(n=1, v=A)
"dA" means Sx receives decided(v=A)
```

- S1 and S2 will reply with `prepare_ok(1, 0, null)` to the `p1` message.
- If `dA` is lost, one of the nodes waiting can run Paxos again and try a new `n` higher than the previous one.

  - the `prepare_ok(2, 1, 'A')` reply will come back,
  - then the node is forced to send `a2vA` and hopefully this time, after the node gets the `accept_ok` message, it will send out `dA` messages that won't get lost again
- a value is said to be chosen when a majority of acceptors in the `accept` handler take the accept branch and accept the value
  - however, not everyone will *know* this, so that's why the `decide` message is sent out

These diagrams are not specific about who the proposer is

- it doesn't really matter
- the proposers are logically separate from the acceptors
- we only care about what acceptors saw and replied

Note Paxos only requires a majority of the servers

- so we can continue even though `S3` was down
- proposer must not wait forever for any acceptor's response

What would happen if network partition?

- I.e. `S3` was alive and had a proposed value B
- `S3` 's prepare would not assemble a majority

The homework question

How does Paxos ensure that the following sequence of events can't happen? What actually happens, and which value is ultimately chosen?

```
proposer 1 crashes after sending two accept() requests
proposer 2 has a different value in mind

A: p1 a1foo
B: p1       p2 a2bar
C: p1 a1foo p2 a2bar

C's prepare_ok to B really included "foo"
  thus a2foo, and so no problem
```

The point:

- if the system has already reached agreement, majority will know value
- any new majority of prepares will intersect that majority
- so subsequent proposer will learn of already-agreed-on value
- and send it in accept msgs

Example 2 (concurrent proposers):

```
A1 starts proposing n=10 by sending prepare(n=10)
A1 sends out just one accept v=10
A3 starts proposing n=11
  but A1 does not receive its proposal
  A3 only has to wait for a majority of proposal responses

A1: p10 a10v10
A2: p10        p11
A3: p10        p11  a11v11

A1 and A3 have accepted different values!
```

What will happen?

- **Q:** What will `A2` do if it gets `a10v10` accept msg from `A1` ?
    - `a10v10` means `accept(n=10,v=10)` which happens after the `prepare->` is sent and the `<-prepare_ok` is received
    - **A:** A2 will reject because it has a higher `np` from `p11`
- **Q:** What will `A1` do if it gets `a11v11` accept msg from `A3` ?
    - **A:** `A1` will reply `ACCEPT_OK` and change its value to 11 because `n = 11 > np = 10`

What if A3 were to crash at this point (and not restart)?

How about this:

```
A1: p10  a10v10              p12
A2: p10          p11  a11v11
A3: p10          p11         p12   a12v10
```

Has the system agreed to a value at this point?

What's the commit point?

- i.e. exactly when has agreement been reached?
- i.e. at what point would changing the value be a disaster?
- after a majority has the same `v_a` ? no -- why not? above counterexample
- after a majority has the same `v_a/n_a` ? yes -- why sufficient? sketch:
    - suppose majority has same `v_a/n_a`
    - acceptors will reject `accept()` with lower `n`
    - for any higher `n` : prepare's must have seen our majority `v_a/n_a` (overlap)
    - what if overlap servers saw `prepare(n)` before `accept(v_a, n_a)` ?
        - would reject `v_a/n_a`
        - thus wouldn't have a majority yet
        - proposer might be free to choose `v != v_a`

Why does the proposer need to pick `v_a` with highest `n_a` ?

```
A1: p10  a10vA              p12
A2: p10          p11  a11vB
A3: p10          p11  a11vB  p12   a12v??

n=11 already agreed on vB
n=12 sees both vA and vB, but must choose vB
```

Two cases:

1. There was a majority before `n=11`
   - `n=11` 's prepares would have seen value and re-used it
   - so it's safe for `n=12` to re-use `n=11` 's value
2. There was not a majority before `n=11`
   - `n=11` might have obtained a majority
   - so it's required for `n=12` to re-use `n=11` 's value

Why does prepare handler check that `n > n_p` ?

- it's taking `max(concurrent n's)` , for accept handler
- responding to all `prepare()` with `prepare_ok()` would be also fine,
  - but proposers with `n < n_p` would be ignored by `accept()` anyway

Why does accept handler check `n >= n_p` ?

- required to ensure agreement
- there's a unique highest `n` active
- everyone favors the highest `n`
- without `n >= n_p` check, you could get this bad scenario:

Scenario:

```
A1: p1 p2 a1vA
A2: p1 p2 a1vA a2vB
A3: p1 p2      a2vB
```

Why does accept handler update `n_p = n` ?

- required to prevent earlier `n` 's from being accepted
- node can get `accept(n,v)` even though it never saw `prepare(n)`
- without `n_p = n` , can get this bad scenario:

Scenario:

```
A1: p1    a2vB a1vA p3 a3vA
A2: p1 p2           p3 a3vA
A3:    p2 a2vB
```

What if new proposer chooses `n < old proposer` ?

- i.e. if clocks are not synced
- cannot make progress, though no correctness problem

What if an acceptor crashes after receiving accept?

```
A1: p1  a1v1
A2: p1  a1v1 reboot  p2  a2v?
A3: p1               p2  a2v?

A2 must remember v_a/n_a across reboot! on disk
  might be only intersection with new proposer's majority
  and thus only evidence that already agreed on v1
```

What if an acceptor reboots after sending `prepare_ok` ?

- does it have to remember `n_p` on disk?
- if `n_p` not remembered, this could happen:

Example:

```
  `S1`: p10            a10v10
  `S2`: p10 p11 reboot a10v10 a11v11
  `S3`:     p11               a11v11
```

- 11's proposer did not see value 10, so 11 proposed its own value
- but just before that, 10 had been chosen!
- b/c `S2` did not remember to ignore `a10v10`

Can Paxos get stuck?

- Yes, if there is not a majority that can communicate
- How about if a majority is available?
  - Possible to livelock: dueling proposers, keep `prepare` 'ing higher `n` 's
    - One reason to try electing a leader: reduce chance of dueling proposers
  - With single proposer and reachable majority, should reach consensus

6.824 2015 Lecture 6: Raft

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

This lecture: Raft

- larger topic is fault tolerance via replicated state machines
- Raft -- a much more complete design than straight Paxos

Raft overview:

```
clients -> leader -> followers -> logs -> execution
```

Raft vs Paxos?

- Our use of Paxos:
  - agrees separately on each client operation
- Raft:
  - agrees on each new leader (and on tail of log)
  - agreement not required for most client operations
  - Raft is Paxos optimized for log appends (more or less)
- why Raft-style leader?
  - no *dueling proposers* (unless leader fails)
    - leader just tells other people what to do
  - fewer messages, less complexity (unless leader fails)
  - well-defined notion of one log being more complete than another
    - simplifies switching leaders (and maybe crash recovery)
    - very hard to find a solution for this in Paxos because logs have "holes"

What about understandability?

- you must decide for yourself
- straight Paxos is simpler than Raft
- but straight Paxos is too simple for practical replication
  - everyone extends it in their own way
  - and ends up with something more or less like Raft
- Paxos+log+leader probably not simpler than Raft

    - though presumably depends on which Paxos variant you choose

Is more direct use of Paxos (like Lab 3) ever a win?

- i.e. is a Raft-style leader ever a bad idea?
- geographically spread peers
- a single leader would be far from some clients
- some peers would be slow to other peers (Paxos tolerates lag)

Let's start w/ Raft w/ no leader change

- for now, reliable leader
- followers may be slow or unreachable (but they do not lose state)
- what do we want?
    1. tolerate a *minority of failed followers*
    2. live followers and dead followers *converge on same log* since replication requires same order of execution
    3. *execute only when entry cannot be lost* (committed) since cannot easily undo execution or reply to client
- idea for ensuring identical log:
    - leader sends *log entry*, *index*, and info about *previous* entry
    - client can reject (e.g I don't have previous entry!)
    - leader backs up for that follower, sends earlier entries
        - leader forces followers' logs to be identical to leader's
- idea for execution:
    - idea #1 means leader knows follower is identical up to some point
    - once a majority are identical up to a point,
        - leader sends that out as commit point,
        - everyone can execute through that point,
        - leader can reply to clients

What to do if the leader crashes?

- other servers time out (no AppendEntries "heart-beats" for a while)
- if other servers are missing heartbeats they start to suspect the leader is down
    - can't really know *for sure* leader is down/up on a network
- choose a new leader!
- Raft divides time into terms
- most terms have a leader

What are the dangers in transition to a new leader?

- two leaders
- no leader
- might forget an executed log entry
- logs might end up different (diverge)

Talk about leader election first, then log consistency at term boundary

How to ensure at most one leader in a term?

- (look at Figure 2, RequestVote RPC, and Rules for Servers)
- leader must get votes from a majority of servers
- **Rule:** server can cast only one vote per term
- thus at most one server can think it has won
- why a majority?
  - the answer is always the same!
  - "requiring a majority means not requiring a minority essentially"
  - allows fault tolerance (failure of minority doesn't impede progress)
  - prevents split brain (at most one candidate can get a majority)
  - ensures overlap (at least one in majority has every previously committed log entry)

Could election fail to choose any leader?

- Yes!
  - > = 3 candidates split the vote evenly or even # of live servers, two candidates each get half

What happens after an election in which no-one gets majority?

- timeout, increment term, new election
- when a server decides it might wants to be a candidate it first waits a random delay and only if it doesn't hear from anyone else then it becomes a candidate
- higher term takes precedence, candidates for older terms quit
- Note: timeout must be longer than it takes to complete election!
- Note: this means some terms may have no leader, no log entries

How does Raft reduce chances of election failure due to split vote?

- each server delays a random amount of time before starting candidacy
- why is the random delay useful?
  - [see diagram of times at which servers' delays expire]

- one will choose lowest random delay
- hopefully enough time to elect before next delay expires
- this idea comes up often in distributed systems

Diagram:

```
          20 ms                  50 ms            80 ms
|-------------*-----------------------*-----------------*-----------|
               S1                      S2                S3
```

How to choose the random delay range?

- too short: 2nd candidate starts before first finishes
- too long: system sits idle for too long after leader fails
- a rough guide:
  - suppose it takes 10ms to complete an unopposed election
  - and there are five servers
  - we want delays to be separated by (say) 20ms
  - so random delay from 0 to 100 ms
  - plus a few multiples of leader heartbeat interval

Remember this random delay idea!

- it's a classic scheme for decentralized soft election; e.g. ethernet

Raft's elections follow a common pattern: separation of safety from progress

- *Hard* mechanisms ensure `< 2` leaders in one term
  - Problem: elections can fail (e.g. 3-way split)
- Solution: always safe to start a new election in a new term
  - Poblem: repeated elections can prevent any work getting done
- Solution: *soft* mechanisms reduce probability of wasted elections
  - heartbeat from leader (remind servers not to start election)
  - timeout period (don't start election too soon)
  - random delays (give one leader time to be elected)

Remember: there's a way to split the problem into "safety/correctness" concerns and "liveness/performance" concerns

What if old leader isn't aware a new one is elected?

- perhaps b/c old leader didn't see election messages
- new leader means a majority of servers have incremented currentTerm
  - so old leader (w/ old term) can't get majority for AppendEntries

- though a minority may accept old server's log entries...
- so logs may diverge at end of old term...

Now let's switch topics to **data handling** at term boundaries

What do we want to ensure?

- each server executes the same client cmds, in the same order
  - i.e. if any server executes, then no server executes something else for that log entry
- as long as single leader, we've already seen it makes logs identical what about when leader changes?

What's the danger?

Leader of term 3 crashed while sending `AppendEntries`

```
S1: 3
S2: 3 3
S3: 3 3
S2 and S3 might have executed; does Raft preserve it?
```

May be a series of crashes, e.g.

```
S1: 3
S2: 3 3 (new leader) 4
S3: 3 3                (new leader) 5
```

Thus different entries for the same index!

Roll-back is a big hammer -- forces leader's log on everyone

- in above examples, whoever is elected imposes log on everyone
- Example:
  - S3 is chosen as new leader for term 6
  - S3 wants to send out a new entry (in term 6)
    - `AppendEntries` says previous entry must have term 5
  - S2 replies false (`AppendEntries` step 2)
  - S3 decrements `nextIndex[S2]`
  - S3 sends `AppendEntries` for the op w/ term=5, saying prev has term=3
  - S2 deletes op from term 4 (`AppendEntries` step 3) and replaces with op for term 5 from S3 (and S1 rejects b/c it doesn't have anything in that entry)
    - S2 sets op for term 6 as well

Ok, leader will force its own log on followers

- but that's not enough!

- can roll-back delete an executed entry?

When is a log entry executed?

- when leader advances `commitIndex/leaderCommit`
- when a majority match the leader up through this point

Could new leader roll back executed entries from end of previous term?

- i.e. could an executed entry be missing from the new leader's log?
- Raft needs to ensure new leader's log contains every potentially executed entry
- i.e. must forbid election of server who might be missing an executed entry

What are the election rules?

- Figure 2 says only vote if candidate's log "at least as up to date"
- So leader will be *at least as up to date* as a majority

What does "at least as up to date" mean?

Could it mean log is >= length? No, example:

```
S1: 5, (leader) 6, (crash + leader) 7,
S2: 5                                  (leader) 8
S3: 5                                           8
```

- first, could this scenario happen? how?
  - S1 leader in epoch 6; crash+reboot; leader in epoch 7; crash and stay down
    - both times it crashed after only appending to its own log
  - S2 leader in epoch 8, only S2+S3 alive, then crash
- who should be next leader?
  - S1 has longest log, but entry 8 is committed !!!
    - Raft adopts leader's log, so S1 as leader -> un-commit entry 8
    - this would be incorrect since S2 may have replied to client
  - so new leader can only be one of S2 or S3
  - i.e. the rule cannot be simply "longest log"

End of 5.4.1 explains "at least as up to date" voting rule

- compare last entry
- higher term wins
- if equal terms, longer log wins

So:

- S1 can't get any vote from S2 or S3, since `7 < 8`
- S1 will vote for either S2 or S3, since `8 > 7`

- S1's operations from terms 6 and 7 will be discarded!
    - ok since no majority -> not executed -> no client reply

The point:

- "at least as up to date" rule causes new leader to have all executed entries in its log
- so new leader won't roll back any executed operation
- similar to Paxos: new round ends up using chosen value (if any) of prev round

The question: Figure 7, which of a/d/f could be elected?

- i.e. majority of votes from "less up to date" servers?

The most subtle thing about Raft (figure 8)

Figure 8:

```
S1 1, L 2,    ,      L 4,
S2 1,   2,    ,      \A/,
S3 1,   <-------- 2 <-| ,
S4 1,    ,    ,         ,
S5 1,    , L 3,         , L will erase all 2's
```

- not 100% true that a log entry on a majority is committed
    - i.e. will never be forgotten
- Figure 8:
    - S1 was leader in term 2, sends out two copies of 2
    - S5 leader in term 3
    - S1 leader in term 4, sends one more copy of 2 (b/c S3 rejected op 4)
    - what if S5 now becomes leader?
        - S5 can get a majority (w/o S1)
        - S5 will roll back 2 and replace it with 3
    - could 2 have executed?
        - it is on a majority...
        - so could S1 have mentioned it in leaderCommit after majority?
        - no! very end of Figure 2 says "log[N].term == currentTerm"
        - and S1 was in term 4 when sending 3rd copy of 2
    - what's Raft's actual commit point?
        - bottom-right of page 310
        - "committed once the leader that created the entry has replicated on majority"
        - and commit point of one entry commits all before it
            - which is how 2 *could* have committed if S1 hadn't lost leadership

Another topic: configuration change (Section 6)

- configuration = set of servers

- how does Raft change the set of servers?
- e.g. every few years might want to retire some, add some
- or move all at once to an entirely new set of server
- or increase/decrease the number of servers

How might a *broken* configuration change work?

- each server has the list of servers in the current config
- change configuation by changing lists, one by one
- example: want to replace S3 with S4
  - S1: 1,2,3 1,2,4
  - S2: 1,2,3 1,2,3
  - S3: 1,2,3 1,2,3
  - S4: 1,2,4 1,2,4
- OOPS!
  - now *two* disjoint group/leaders can form:
    - S2 and S3 (who know nothing of new config)
    - S1 and S4
  - both can process client requests, so split brain

Raft configuration change

- **Idea:** "join consensus" stage that includes *both* old and new configuration
- leader of old group logs entry that switches to joint consensus
  - during joint consensus, leader separately logs in old and new
    - i.e. *two* log and *two* agreements on each log entry
    - this will force new servers to catch up and force new and old logs to be the same
- after majority of old and new have switched to joint consensus,
  - leader logs entry that switches to final configuration

Example (which won't make sense because it's not properly illustrated in the original notes):

```
S1: 1,2,3  1,2,3+1,2,4
S2: 1,2,3
S3: 1,2,3
S4:        1,2,3+1,2,4
```

- if crash but new leader didn't see the switch to joint consensus,
  - then old group will continue, no switch, but that's OK
- if crash and new leader did see the switch to joint consensus,
  - it will complete the configuration change

Performance

- no numbers on how fast it can process requests
- what are the bottlenecks likely to be?
- disk:
  - need to write disk for client data durability, and for protocol promises
  - write per client request? so 100 per second?
  - could probably batch and get 10,000 to 100,000
- net: a few message exchanges per client request
  - 10s of microseconds for local LAN message exchange?
  - so 100,000 per second?

Next week: use of a Raft-like protocol in a complex application

Russ Cox's lecture on Go

Why Go?

- an answer to the problems of scalability at Google
    - `10^6+` machines design point
    - it's routine to be running on 1000 machines
    - constantly writing programs that coordinate with each other
        - sometimes MapReduce works, other times it doesn't

Who uses Go at Google

- SPDY proxy for Chrome on mobile devices uses a Go-written *Data Compression Proxy*
- dl.google.com
- YouTube MySQL balancer
- the target is network servers, but it's a great gen. purp. language
- Bitbucket, bitly, GitHub, Dropbox, MongoDB, Mozilla services, NY Times, etc.

Concurrency

- "Communicating Sequential Processes", by Hoare, 1978
    - strongly encouraged to read
    - in some sense, a generalization of UNIX pipelines
- Bell Labs had some languages developed for concurrency in 80's, 90's:
    - Pan, Promela, Newsqueak, Alef, Limbo, Libthread, Concurrent ML
- Google developed Go in the 2000s

There's no goroutine IDs

- "There's no goroutine IDs, so I can't kill my threads"
    - This is what channels are for: just tell your thread via a channel to shut itself off
    - Also, it's kind of "antisocial" to kill them.
        - What we mean is that your program is prolly not gonna work very well if you keep killing your threads like that

Channels vs. Mutexes

- if you need a mutex, use a mutex
- if you need condition variable, think about using a channel instead
- don't communicate by sharing memory, you share memory by communicating

Network channels

- it'd be great to have the equivalent for a network channel
- if you take local abstractions (like channels) and use them in a new context like a network, ignoring failure modes (etc), then you're gonna run into trouble

Scale of engineering efforts

In 2011, Google had:

- 5000+ developers
- 20+ changes per minute
- 50% code base changes every month (files? not lines probably)
- 50 million test cases executed per day
- single code tree projects

A new language was needed to fix the problems that other languages had with software engineering at this scale

The scale of compilation matters.

- When you compile a package A that depends on B, most (all?) languages need to compile B first
- Go doesn't.
- Dependencies like these at the scale of Google projects slow down compilation if you use a traditional language
    - gets worse with "deeper" dependencies `A->B->C->D->...`
- *Example:* at some point they found a postscript interpreter compiled in a server binary for no reason due to weird deps

Interfaces vs. inheritance

- inhertance hierarchies are hard to get right and if you don't they are hard to change later
- interfaces are much more informal and clearer about who owns and supplies what parts of the program

Readability and simplicity

- Dick Gabriel quote:
  > "I'm always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?"
- Simplify syntax
- Avoid cleverness: ternary operators, macros
- Don't let code writing be like "arguing with your compiler"
- Don't want to puzzle through code 6 months later

Design criteria

- started by Rob Pike, Robert Griesemer and Ken Thompson in late 2007
- Russ Cox, Ian Lance Taylor joined in mid-2008
- design by consensus (everyone could veto a feature, if they didn't want it)

Generics

- Russ: "Don't use `*list.List`, you almost never need them. Use slices."
  - Generics are not bad, just hard to do right.
    - Early designers for Java generics also agreed and warned Go designers to be careful
      - Seems like they regretted getting into that business

Enginering tools

- when you have millions of lines of code, you need mechanical help
  - like changing an API
- Go designed to be easy to parse (not like C++)
- standard formatter
- Means you can't tell a mechanical change from a manual change
  - enables automated rewrites of code

More automation

- fix code for API updates
  - early Go versions API changed a lot
  - Google had a rewriter that would fix your code which used the changed APIs

- renaming struct fields, variables w/ conflict resolution
- moving packages
- splitting of packages
- code cleanup
- change C code to Go
- global analysis that figure out what are all the implementors of an interface for instance

State of Go

- Go 1.4 released in Decembeer 2014
- Go 1.5 has toolchain implemented in Go, not in C
    - concurrent GC
    - Go for mobile devices
    - Go on PowerPC, ARM64
- Lots of people use it
- Go conferences outside of Google/Go

Q&A

- Go vs C/C++
    - Go is garbage collected, biggest difference, so slower
    - Go can be faster than Java sometimes
    - once you're aware of that, you can write code that runs faster than C/C++ code
    - no reason that code that doesn't allocate memory shouldn't run as fast as C/C++
- Goal to use Go outside Google?
    - Yes! Otherwise the language would die?
    - You get a breadth of experts that give you advice and write tools, etc.
        - C++ memory model guy gave feedback on Go memory model
            - Very usefl
    - Not trying to replace anything like language X
        - but they were using C/C++ and didn't want to anymore
        - however Python and Ruby users are switching to Go more
            - Go feels just as light but statically type checked
- Studies about benefits of Go?
    - not a lot of data collected

6.824 2015 Lecture 8: Harp

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Paper: Replication in the Harp File System

- Liskov, Ghemawat, Gruber, Johnson, Shrira, Williams
- SOSP 1991

Why are we reading this paper?

- Harp was the first complete primary/backup system that dealt w/ partition
- It's a complete case study of a replicated service (file server)
- It uses Raft-like replication techniques

How could a 1991 paper still be worth reading?

- Harp introduced techniques that are still widely used
- There are few papers describing complete replicated systems

The paper is a mix of fundamentals and incidentals

- We care a lot about replication
- We may not care much about NFS specifically
  - But we care a lot about the challenges faced when integrating a real application with a replication protocol.
- And we care about where optimization is possible.

I'm going to focus on parts of Harp that aren't already present in Raft.

- But note that Harp pre-dates Raft by 20+ years.
- Raft is, to a great extent, a tutorial on ideas pioneered by Harp.
  - Though they differ in many details.

What does Harp paper explain that Raft paper does not?

- Adapting a complex service to state machine abstraction
  - e.g. possibility of applying an operation twice
- Lots of optimizations
  - pipelining of requests to backup
  - witness, to reduce the # of replicas
  - primary-only execution of read-only operations using leases

- Efficient re-integration of re-started server with large state
  - Don't want to do things like copying entire disks
  - "Catch-up" for re-joining replicas
- Power failure, including simultaneous failure of all servers
- Efficient persistence on disk
- Freeing of log

Harp authors had not implemented recovery yet

- Earlier paper (1988) describes View-stamped Replication
- Later (2006) paper has clearer description, though a bit different:

Basic setup is familiar

- Clients, primary, backup(s), witness(es).
- Client -> Primary
- Primary -> Backups
- Backups -> Primary
  - Primary waits for all backups / promoted witnesses in current view
  - Commit point
- Primary -> execute and reply to Client
- Primary -> tell Backups to Commit
- `2n+1` servers, `n` backups, `n` witnesses, 1 primary
  - need a majority of `n+1` servers `=>`
  - tolerate up to `n` failures
- clients send NFS requests to primary
  - primary forwards each request to all the backups
  - after all the backups have replied, the primary can execute the op and apply it to its FS
  - in a later operation, primary piggybacks an ACK to tell backups the op. commited

Why are `2b+1` servers necessary to tolerate `b` failures?

- (This is review...)
- Suppose we have `N` servers, and execute a write.
- Can't wait for more than `N-b`, since `b` might be dead.
- So let's require waiting for `N-b` for each operation.
- The `b` we didn't wait for might be live and in another partition.
- We can prevent them from proceeding if `N-b > b`.
- I.e. `N > 2b => N = 2b + 1` is enough.

What are Harp's witnesses?

- The primary and backups have FSs
- The witnesses don't receive anything from primary and don't have FSs
- Suppose we have a `P, B` and a `W`
- If there's a partition `P | B, W`, the witness acts as a tie breaker
  - whichever one (P or B) can talk to the witness gets to continue and execute client side operations
  - witness acts as a tie breaker: whoever can talk to it wins and gets to act as a primary
- a second use of the witness is to record operations
- once a witness is part of the partition `B, W`, it records operations so that a majority of nodes have the latest operations
- a final function of the witness is that when the primary comes back to life, the witness has been logging every single operation issued since primary disappeared, so witness can replay every op to primary and primary will be up to date w.r.t. all the operations executed
  - efficiently bring primary up to speed
  - the backup could do that too, but Harp is designed so that backup dumps the op logs to disk and witnesses keep the logs themselves so they can quickly send them to primary for reapplying
    - assumption that reapplying witness logs is faster than copying backup disk
    - assumption that witness logs won't get too big
- The witnesses are one significant difference from Raft.
- The `b` witnesses do not ordinarily hear about operations or keep state.
- Why is that OK?
  - `b+1` of `2b+1` do have state
  - So any `b` failures leaves at least one live copy of state.
- Why are the `b` witnesses needed at all?
  - If `b` replicas with state do fail, witnesses give the required `b+1` majority.
  - To ensure that only one partition operates -- no split brain.
- So, for a 3-server system, the witness is there to break ties about which partition is allowed to operate when primary and backup are in different partitions.
  - The partition with the witness wins.

Does primary need to send operations to witnesses?

- The primary must collect ACKs from a majority of the `2b+1` for every r/w operation.
  - To ensure that it is still the primary -- still in the majority partition.
  - To ensure that operation is on enough servers to intersect with any
    - future majority that forms a new view.
- If all backups are up, primary+backups are enough for that majority.
- If `m` backups are down:

    - Primary must talk to `m` "promoted" witnesses to get its majority for each op.
    - Those witnesses must record the op, to ensure overlap with any
        - future majority.
    - Thus each "promoted" witness keeps a log.
- So in a `2b+1` system, a view always has `b+1` servers that the primary must contact for each op, and that store each op.

Note: somewhat different from Raft

- Raft keeps sending each op to all servers, proceeds when majority answer
    - So leader must keep full log until failed server re-joins
- Harp eliminates failed server from view, doesn't send to it
    - Only witness has to keep a big log; has special plan (ram, disk, tape).
- The bigger issue is that it can take a lot of work to bring a re-joining replica up to date; careful design is required.

What's the story about the UPS?

- This is one of the most interesting aspects of Harp's design
- Each server's power cord is plugged into a UPS
- UPS has enough battery to run server for a few minutes
- UPS tells server (via serial port) when main A/C power fails
- Server writes dirty FS blocks and Harp log to disk, then shuts down

What does the UPS buy for Harp?

- Efficient protection against A/C power failure of ALL servers
- For failures of up to b servers, replication is enough
- If *all* servers failed and lost state, that's more than b failures,
    - so Harp has no guarantee (and indeed no state!)
- With UPS:
    - Each server can reply *without* writing disk!
    - But still guarantees to retain latest state despite simultaneous power fail
- But note:
    - UPS does not protect against other causes of simultaneous failure
    - e.g. bugs, earthquake
    - Harp treats servers that re-start after UPS-protected crash differently
        - than those that re-start with crash that lost in-memory state
    - Because the latter may have forgotten *committed* operations
- For independent failures Harp has powerful guarantees, for stuff like software bugs that will cause a cascade of crashes, it doesn't really have solutions

Larger point, faced by every fault-tolerant system

- Every replicated system tends to need to have a commit point
- Replicas *must* keep persistent state to deal w/ failure of all servers
    - Committed operations
    - Latest view number, proposal number, &c
- Must persist this state before replying
- Writing every commit to disk is very slow!
    - 10 ms per disk write, so only 100 ops/second
- So there are a few common patterns:
    1. Low throughput
    2. Batching, high delay
        - batch a lot of writes and do them all at the same time to amortize the cost of each write
        - but now you need to make clients wait for their write to finish more
            - because they are also waiting for other clients' writes to finish
    3. Lossy or inconsistent recovery from simultaneous failure
        - no guarantees after crashes
    4. Batteries, flash, SSD w/ capacitor, &c

Let's talk about Harp's log management and operation execution

- Primary and backup must apply client operations to their state
- State here is a file system -- directories, file, owners, permissions, &c
- Harp must mimic an ordinary NFS server to the client
    - i.e. not forget about ops for which it has sent a reply

What is in a typical log record?

- Not just the client-issued op., like `chmod`

Log record stores:

- Client's NFS operation (write, mkdir, chmod, &c)
- Shadow state: modified i-nodes and directory content *after* execution
    - (i.e. the results after executing the operation)
- Client RPC request ID, for duplicate detection
    - Primary might repeat an RPC if it thinks the backup has failed
- Reply to send to client, for duplicate detection

Why does Harp have so many log pointers?

- FP most recent client request
- CP commit point (real in primary, latest heard in backup)
- AP highest update sent to disk

- LB disk has finished writing up to here
- GLB all nodes have completed disk up to here

Why the FP-CP gap?

- So primary doesn't need to wait for ACKs from each backup
  - before sending next operation to backups
- Primary pipelines ops CP..FP to the backups.
- Higher throughput if concurrent client requests.

Why the AP-LB gap?

- Allows Harp to issue many ops as disk writes before waiting for disk
- The disk is more efficient if it has lots of writes (e.g. arm scheduling)

What is the LB?

- This replica has everything `<= LB` on disk.
- So it won't need those log records again.

Why the LB-GLB gap?

- GLB is min(all servers' LBs).
- GLB is earliest record that *some* server might need if it loses memory.

When does Harp execute a client operation?

There are two answers!

1. When operation arrives, primary figures out exactly what should happen.
   - Produces resulting on-disk bytes for modified i-nodes, directories, &c.
   - This is the shadow state.
   - This happens before the CP, so the primary must consult recent operations in the log to find the latest file system state.
2. After the operation commits, primary and backup can apply it to their file systems.
   - They copy the log entry's shadow state to the file system;
   - they do not really execute the operation.
   - And now the primary can reply to the client's RPC.

Why does Harp split execution in this way?

- If a server crashes and reboots, it is brought up to date by replaying log entries it might have missed. Harp can't know exactly what the last pre-crash operation was, so Harp may repeat some. It's not correct to fully execute some operations twice, e.g. file append.

- So Harp log entries contain the *resulting* state, which is what's applied.

Append example:

```
                         /---> picks up the modified inode
                        /
--------------------------------
     |   Append1   |  Append2  |
---- * ------------------------
    / \        \
     |          \-> new inode stored here
    CP
```

If backup crashes after it writes A1 to disk but before replying to primary, when the backup reboots there's no obvious way of telling whether it executed A1. As a result, it has to reexecute it. Thus, these log records have to be "repeatable."

Actually, a lot of replication systems have to cope with this, and this is one way to deal with it. It also illustrates how non-straightforward replication can be.

- Harp needs to be aware of FS-level inodes for instance

The point: multiple replay means replication isn't transparent to the service.

- Service must be modified to generate and accept the state modifications that result from client operations.
- In general, when applying replication to existing services, the service must be modified to cope with multiple replay.

Can Harp primary execute read-only operations w/o replicating to backups?

- e.g. reading a file.
- Would be faster -- after all, there's no new data to replicate.
- The reason we forward read only operations to backup is to make sure we find out if we were partitioned and 1000 ops were executed that we don't know about: make sure we don't reply with an old write of the data we are reading
- What's the danger?
- Harp's idea: leases
- Backups promise not to form a new view for some time.
  - (i.e. not to process any ops as a primary)
- Primary can execute read-only ops locally for that time minus slop.
  - because it knows backup won't execute ops as primary during that time (backup promised this!)
- Depends on reasonably *synchronized clocks*:
  - Robert Morris: "Not really happy about this."
  - Primary and backup must have bounded disagreement on how fast time passes.

    - This really requires a bounded frequency skew
        - Apparently hardware is really bad at providing this

What state should primary use when executing a read-only operation?

- Does it have to wait for all previously arrived operations to commit?
    - No! That would be almost as slow as committing the read-only op.
- Should it look at state as of operation at FP, i.e. latest r/w operation?
    - No! That operation has not committed; not allowed to reveal its effects.
- Thus Harp executes read-only ops with state as of CP.
- What if client sends a WRITE and (before WRITE finishes) a READ of same data?
    - READ may see data *before* the WRITE!
    - Why is that OK?
    - The client sent the READ and the WRITE concurrently. It has no right to expect one order or the other. So if the READ doesn't see the WRITE's effects, that's acceptable -- it's the same answer you'd get if the READ had moved through the network faster than the WRITE, which could happen. You can only expect a READ to see a WRITE's effect if you issue the WRITE, wait for a reply to the WRITE, and then issue the READ.

How does failure recovery work?

- I.e. how does Harp recover replicated state during view change?

Setup for the following scenarios:

5 servers: S1 is usually primary, S2+S3 are backups, S4+S5 witnesses

Scenario:

```
S1+S2+S3; then S1 crashes
S2 is primary in new view (and S4 is promoted)
```

- Will S2 have every committed operation?
    - Yes.
- Will S2 have every operation S1 received?
    - No. No, maybe op didn't reach S2 from S1 and then S1 crashed.
- Will S2's log tail be the same as S3's log tail?
    - Not necessarily.
        - Maybe op reached S2 but not S3 and then S1 crashed.
        - Maybe op reached S2, and S3 crashed, so S4 was promoted. Then S3 came back up?
- How far back can S2 and S3 log tail differ?
    - Not up to the CP, because committed ops could be committed w/ help of promoted

witness `=>` backup logs differ

- How to cause S2 and S3's log to be the same?
  - Must commit ops that appeared in both S2+S3 logs
  - What about ops that appear in only one log?
    - In this scenario, can discard since could not have committed
    - But in general committed op might be visible in just one log
- From what point does promoted witness have to start keeping a log?

What if S1 crashed just before replying to a client?

- Will the client ever get a reply?

After S1 recovers, with intact disk, but lost memory.

- It will be primary, but Harp can't immediately use its state or log.
  - Unlike Raft, where leader only elected if it has the best log.
  - Harp must replay log from promoted witness (S4)
- Could S1 have executed an op just before crashing that the replicas didn't execute after taking over?
  - No, execution up to CP only, and CP is safe on S2+S3.

New scenario: S2 and S3 are partitioned (but still alive)

- Can S1+S4+S5 continue to process operations?
  - Yes, promoted witnesses S4+S5
- S4 moves to S2/S3 partition
- Can S1+S5 continue?
  - No, primary S1 doesn't get enough backup ACKs
- Can S2+S3+S4 continue?
  - Yes, new view copies log entries S4->S2, S4->S3, now S2 is primary
- Note:
  - New primary was missing many committed operations
  - In general some *committed* operations may be on only one server

New scenario: S2 and S3 are partitioned (but still alive)

- S4 crashes, loses memory contents, reboots in S2/S3 partition
- Can they continue?
  - Only if there wasn't another view that formed and committed more ops
- How to detect?
  - Depends on what S4's on-disk view # says.
  - OK if S4's disk view # is same as S2+S3's.
    - No new views formed.
    - S2+S3 must have heard about all committed ops in old view.

Everybody (S1-5) suffers a power failure.

- S4 disk and memory are lost, but it does re-start after repair.
- S1 and S5 never recover.
- S2 and S3 save everything on disk, re-start just fine.
- Can S2+S3+S4 continue?
- (harder than it looks)
    - Believe the answer is "No": cannot be sure what state S4 had before failure.
    - Might have formed a new view with S1+S5, and committed some ops.
        - Can S2 and S3 know about the previous view? Not always.

When can Harp form a new view?

1. No other view possible.
2. Know view # of most recent view.
3. Know all ops from most recent view.

Details:

- (1) is true if you have n+1 nodes in new view.
- (2) true if you have n+1 nodes that did not lose view # since last view.
    - View # stored on disk, so they just have to know disk is OK.
    - One of them *must* have been in the previous view.
    - So just take the highest view number.
- And #3?
    - Need a disk image, and a log, that together reflect all operations through the end of the previous view.
    - Perhaps from different servers, e.g. log from promoted witness, disk from backup that failed multiple views ago.

Does Harp have performance benefits?

- In Fig 5-1, why is Harp *faster* than non-replicated server?
- How much win would we expect by substituting RPC for disk operations?

Why graph x=load y=response-time?

- Why does this graph make sense?
- Why not just graph total time to perform X operations?
- One reason is that systems sometimes get more/less efficient w/ high load.
- And we care a lot how they perform w/ overload.

Why does response time go up with load?

- Why first gradual...

    - Queuing and random bursts?
    - And some ops more expensive than others, cause temp delays.
- Then almost straight up?
    - Probably has hard limits, like disk I/Os per second.
    - Queue length diverges once offered load > capacity

6.824 2015 Lecture 9: DSM and Sequential Consistency

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Topic: Distributed computing

- parallel computing on distributed machines
- 4 papers on how to use a collection of machines to solve big computational problems
  - we already read one of them: MapReduce
- 3 other papers (IVY, TreadMarks, and Spark)
  - two provide a general-purpose memory model
  - Spark is in MapReduce style

Distributed Shared Memory (DSM) programming model

- Adopt the same programming model that multiprocessors offer
- Programmers can use locks and shared memory
  - Programmers are familiar with this model
  - e.g., like goroutines sharing memory
- General purpose
  - e.g., no restrictions like with MapReduce
- Applications that run on a multiprocessor can run on IVY/TreadMarks

Challenge: distributed systems don't have physical shared memory

- On a network of cheap machines
  - [diagram: LAN, machines w/ RAM, MGR]

Diagram:

```
    *----------*   *----------*   *----------*
    |          |   |          |   |          |
    |          |   |          |   |          |
    |          |   |          |   |          |
    *-----*----*   *-----*----*   *-----*----*
          |              |              |
  --------*--------------*--------------*--------  LAN
```

Diagram:

```
          M0           M1
    *----------*   *----------*
    | M0 acces |   | x x x x  |
    |----------|   |----------|
    | x x x x  |   | M1 acces |
    *-----*----*   *-----*----*
          |              |
  --------*--------------*------- LAN

  The 'xxxxx' pages are not accesible locally,
  they have to be fetched via the network
```

Approach:

- Simulate shared memory using hardware support for virtual memory
- General idea illustrated with 2 machines:
    - Part of the address space maps a part of M0's physical memory
        - On M0 it maps to the M0's physical page
        - On M1 it maps to not present
    - Part of the address space maps a part of M1's physical memory
        - On M0 it maps to not present
        - On M1 it maps to its physical memory
- A thread of the application on M1 may refer to an address that lives on M0
    - If thread LD/ST to that "shared" address, M1's hardware will take a page fault
        - Because page is mapped as not present
    - OS propagates page fault to DSM runtime
    - DSM runtime can fetch page from M0
    - DSM on M0, maps page not present, and sends page to M1
    - DSM on M1 receives it from M0, copies it somewhere in memory (say address 4096)
    - DSM on M1 maps the shared address to physical address 4096
    - DSM returns from page fault handler
    - Hardware retries LD/ST
- Runs threaded code w/o modification
    - e.g. matrix multiply, physical simulation, sort

Challenges:

- How to implement it efficiently?
    - IVY and Treadmarks
- How to provide fault tolerance?
    - Many DSMs punt on this
    - Some DSM checkpoint the whole memory periodically
    - We will return to this when talking about Spark

Correctness: coherence

- We need to articulate what is correctness before optimizing performance
  - Optimizations should preserve correctness
- Less obvious than it may seem!
  - Choice trades off between performance and programmer-friendliness
  - Huge factor in many designs
  - More in next lecture
- Today's paper assumes a simple model
  - The distributed memory should behave like a single memory
  - Load/stores much like put/gets in labs 2-4

Example 1:

```
x and y start out = 0

M0:
  x = 1
  if y == 0:
    print yes
M1:
  y = 1
  if x == 0:
    print yes

Can they both print "yes"?
```

Naive distributed shared memory

Diagram 1:

- M0, M1, M2, LAN
- each machine has a local copy of all of memory
- read: from local memory
- write: send update msg to each other host (but don't wait)
- fast: never waits for communication

Does this naive memory work well?

- What will it do with Example 1?
  - It can fail because M0 and M1 could not see the writes by the time their `if` statements are reached so they will both print *yes*.
- Naive distributed memory is fast but incorrect

Diagram (broken scheme):

```
       M0
  *----------*   *----------*   *----------*
  |          |   |          |   |          |
  |        ------------------------> wAx   |
  |        ----------> wAx  |   |          |
  *-----*----*   *-----*----*   *-----*----*
        |              |              |
--------*--------------*--------------*-------- LAN
```

- M0 does write locally and tells other machines about the write after it has done it
- imagine what output you would get instead of 9, if each machine was running a program that incremented the value at address A 3 times

Coherence = *sequential consistency*

- "Read sees *most recent* write" is not clear enough when you have multiple processes
- Need to nail down correctness a bit more precisely
- Sequential consistency means:
  - The result of any execution is the same as if
    - the operations of all the processors were executed in some sequential order (total order)
    - and the operations of each individual processor appear in this sequence in the order specified by its program
      - (if P says A before B, you can't have B; A; show up in that seq. order)
    - and read sees last write in total order
- There must be some total order of operations such that
  1. Each machine's instructions appear in-order in the order
  2. All machines see results consistent with that order
     - i.e. reads see most recent write in the order
- Behavior of a single shared memory

Would sequential consistency cause our example to get the intuitive result?

Sequence:

```
M0: Wx1 Ry?
M1: Wy1 Rx?
```

- The system is required to merge these into one order, and to maintain the order of each machine's operations.
- So there are a few possibilities:
  - `Wx1 Ry0 Wy1 Rx1`
  - `Wx1 Wy1 Ry1 Rx1`
  - `Wx1 Wy1 Rx1 Ry1`
  - others too, but all symmetric?

- What is forbidden?
  - `Wx1 Ry0 Wy1 Rx0` -- Rx0 read didn't see preceding Wx1 write (naive system did this)
  - `Ry0 Wy1 Rx0 Wx1` -- M0's instructions out of order (some CPUs do this)

Go's memory consistency model

- What is Go's semantics for the example?
- Go would allow both goroutines to print "yes"!
- Go race detector wouldn't like the example program anyway
- Programmer is *required* to use locks/channels to get sensible semantics
- Go doesn't require the hardware/DSM to implement strict consistency
- More about weaker consistency on Thursday

Example:

```
x = 1
y = 2
```

- Go's memory model tells you if thread A will see the write to x if it has seen the write to y
  - In Go, there's no guarantee x's write will be seen if y was written

A simple implementation of sequential consistency

A straightforward way to get sequential consistency: Just have a manager in between the two or three machines that interleaves their instructions

```
      *----------*   *----------*
      |          |   |          |
      |          |   |          |
      |          |   |          |
      *-----*----*   *-----*----*
            |              |
    --------*--------------*--------
                   |
                   |
             *----------*
             | inter-   |
             | leaver   |
             |          |
             *-----*----*
                   |
                  -*-
                  \ /
                   .
                  RAM
```

Diagram 2:

```
  *----------*   *----------*
  |          |   |          |
  |          |   |          |
  |          |   |          |
  *-----*----*   *-----*----*
        |              |
--------*--------------*--------
               |
               |
         *----------*
         |          |
         |          |
         |          |
         *-----*----*
               |
              -*-
              \ /
               .
              RAM
```

- single memory server
- each machine sends r/w ops to server, in order, waiting for reply
- server picks order among waiting ops
- server executes one by one, sending replies
- big ideas:
  - if people just read some data, we can replicate it on all of them
  - if someone writes data, we need to prevent other people from writing it
    - so we take the page out of those other people's memory

This simple implementation will be slow

- single server will get overloaded
- no local cache, so all operations wait for server

Which brings us to **IVY**

- IVY: Integrated shared Virtual memory at Yale
- Memory Coherence in Shared Virtual Memory Systems, Li and Hudak, PODC 1986

IVY big picture

```
[diagram: M0 w/ a few pages of mem, M1 w/ a few pages, LAN]
```

- Operates on pages of memory, stored in machine DRAM (no mem server)
- Each page present in each machine's virtual address space
- On each a machine, a page might be invalid, read-only, or read-write
- Uses VM hardware to intercept reads/writes

Invariant:

- A page is either:

    - Read/write on one machine, invalid on all others; or
    - Read/only on ≥ 1 machines, read/write on none
- Read fault on an invalid page:
    - Demote R/W (if any) to R/O
    - Copy page
    - Mark local copy R/O
- Write fault on an R/O page:
    - Invalidate all copies
    - Mark local copy R/W

IVY allows multiple reader copies between writes

- For speed -- local reads are fast
- No need to force an order for reads that occur between two writes
- Let them occur concurrently -- a copy of the page at each reader

Why crucial to invalidate all copies before write?

- Once a write completes, all subsequent reads *must* see new data
- Otherwise we break our example, and don't get sequential consistency

How does IVY do on the example?

- I.e. could both M0 and M1 print "yes"?
- If M0 sees y == 0,
    - M1 hasn't done ites write to y (no stale data == reads see prior writes),
    - M1 hasn't read x (each machine in order),
    - M1 must see x == 1 (no stale data == reads see prior writes).

Message types:

- [don't list these on board, just for reference]
- RQ read query (reader to manager (MGR))
- RF read forward (MGR to owner)
- RD read data (owner to reader)
- RC read confirm (reader to MGR)
- &c

(see ivy-code.txt on web site)

Scenario 1: M0 has writeable copy, M1 wants to read

Diagram 3:

```
[time diagram: M 0 1]
```

1. M1 tries to read gets a page fault
    - b.c. page must have been marked invalid since M0 has it for R/W (see invariant described earlier)
2. M1 sends RQ to MGR
3. MGR sends RF to M0, MGR adds M1 to `copy_set`
    - What is `copy_set`?
    - "The `copy_set` field lists all processors that have copies of the page. This allows an invalidation operation to be performed without using broadcast."
4. M0 marks page as $access = read$, sends RD to M1
5. M1 marks $access = read$, sends RC to MGR

Scenario 2: now M2 wants to write

Diagram 4:

```
[time diagram: M 0 1 2]
```

1. Page fault on M2
2. M2 sends WQ to MGR
3. MGR sends IV to copy_set (i.e. M1)
4. M1 sends IC msg to MGR
5. MGR sends WF to M0, sets owner=M2, copy_set={}
6. M0 sends WD to M2, access=none
7. M2 marks r/w, sends WC to MGR

Q: What if two machines want to write the same page at the same time?

Q: What if one machine reads just as ownership is changing hands?

Does IVY provide strict consistency?

- no: MGR might process two STs in order opposite to issue time
- no: ST may take a long time to revoke read access on other machines
    - so LDs may get old data long after the ST issues

What if there were no IC message?

TODO: What is IC?

- (this is the new Question)
- i.e. MGR didn't wait for holders of copies to ACK?

No WC?

TODO: What is WC?

- (this used to be The Question)
- e.g. MGR unlocked after sending WF to M0?
- MGR would send subsequent RF, WF to M2 (new owner)
- What if such a WF/RF arrived at M2 before WD?
    - No problem! M2 has `ptable[p].lock` locked until it gets WD
- RC + `info[p].lock` prevents RF from being overtaken by a WF
- so it's not clear why WC is needed!
    - but I am not confident in this conclusion

What if there were no RC message?

- i.e. MGR unlocked after sending RF?
- could RF be overtaken by subsequent WF?
- or by a subsequent IV?

In what situations will IVY perform well?

1. Page read by many machines, written by none
2. Page written by just one machine at a time, not used at all by others

Cool that IVY moves pages around in response to changing use patterns

Will page size of e.g. 4096 bytes be good or bad?

- good if spatial locality, i.e. program looks at large blocks of data
- bad if program writes just a few bytes in a page
    - subsequent readers copy whole page just to get a few new bytes
- bad if false sharing
    - i.e. two unrelated variables on the same page
        - and at least one is frequently written
    - page will bounce between different machines
        - even read-only users of a non-changing variable will get invalidations
    - even though those computers never use the same location

What about IVY's performance?

- after all, the point was speedup via parallelism

What's the best we could hope for in terms of performance?

- $N \times$ faster on N machines

What might prevent us from getting $N \times$ speedup?

- Application is inherently non-scalable
    - Can't be split into parallel activities

- Application communicates too many bytes
  - So network prevents more machines yielding more performance
- Too many small reads/writes to shared pages
  - Even if # bytes is small, IVY makes this expensive

How well do they do?

- Figure 4: near-linear for PDE (partial derivative equations)
- Figure 6: very sub-linear for sort
  - sorting a huge array involves moving a lot of data
  - almost certain to move all data over the network at least once
- Figure 7: near-linear for matrix multiply
- in general, you end up being limited by network throughput for instance when reading a lot of pages

Why did sort do poorly?

- Here's my guess
- N machines, data in 2*N partitions
- Phase 1: Local sort of 2*N partitions for N machines
- Phase 2: 2N-1 merge-splits; each round sends all data over network
- Phase 1 probably gets linear speedup
- Phase 2 probably does not -- limited by LAN speed
  - also more machines may mean more rounds
- So for small # machines, local sort dominates, more machines helps
- For large # machines, communication dominates, more machines don't help
- Also, more machines shifts from n*log(n) local sort to n^2 bubble-ish short

How could one speed up IVY?

- next lecture: relax the consistency model
  - allow multiple writers to same page!

Paper intro says DSM subsumes RPC -- is that true?

- When would DSM be better than RPC?
  - More transparent. Easier to program.
- When would RPC be better?
  - Isolation. Control over communication. Tolerate latency.
  - Portability. Define your own semantics.
  - Abstraction?
- Might you still want RPC in your DSM system? For efficient sleep/wakeup?

Known problems in Section 3.1 pseudo-code

- Fault handlers must wait for owner to send `p` before confirming to manager
- Deadlock if owner has page R/O and takes write fault
  - Worrisome that no clear order `ptable[p].lock` vs `info[p].lock`
  - TODO: Whaaaat?
- Write server / manager must set `owner = request_node`
- Manager parts of fault handlers don't ask owner for the page
- Does processing of the invalidate request hold `ptable[p].lock?`
  - probably can't -- deadlock

6.824 2015 Lecture 10: Consistency

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Today: consistency

- Lazy release consistency
- Using lazy consistency to get performance
- Consistency = meaning of concurrent reads and writes
- Less obvious than it may seem!
- Choice trades off between performance and programmer-friendliness
    - Huge factor in many designs
- Today's paper: a case study

Many systems have storage/memory w/ concurrent readers and writers

- Multiprocessors, databases, AFS, labs
- You often want to improve in ways that risk changing behavior:
    - add caching
    - split over multiple servers
    - replicate for fault tolerance
- How do we know if an optimization is correct?
- We need a way to think about correct execution of distributed programs
- Most of these ideas from multiprocessors and databases 20/30 years ago

How can we write correct distributed programs w/ shared storage?

- Memory system promises to behave according to certain rules
- We write programs assuming those rules
- Rules are a "consistency model"
- Contract between memory system and programmer

What makes a good consistency model?

- There are no "right" or "wrong" models
- A model may make it harder or easier to program
    - i.e. lead to more or less intuitive results
- A model may be harder or easier to implement efficiently
- Also application dependent
    - e.g. Web pages vs memory

Some consistency models:

- Spanner: external consistency (behaves like a single machine)
- Database world: strict serializability, serializability, snap-shot isolation, read-committed
- Distributed file systems: open-to-close consistency
- Computer architects: TSO (total store ordering), release consistency, etc.
- Concurrency theory: sequential consistency, linearizability
- Similar ideas, but sometimes slightly different meaning

DSM is a good place to start to study consistency

- Simple interface: read and write of memory locations
- Consistency well developed in architecture community

Example:

```
x and y start out = 0
M0:
  x = 1
  if y == 0:
    print yes
M1:
  y = 1
  if x == 0:
    print yes
Can they both print "yes"?
```

Performance of DSM is limited by memory consistency

- With sequential consistency, M0's write must be visible to M1 before M0 can execute read
    - Otherwise both M0 and M1 can read 0 and print "yes"
    - (Second "forbidden" example)
- Thus operations will take a while in a distributed system
    - And they have to be done one by one

Treadmarks high level goals?

- Better DSM performance
- Run existing parallel code (multithreaded)
    - this code already has locks
    - TreadMarks will run each thread/process on a separate machine and let it access the DSM.
    - TreadMarks takes advantage that the code already uses locking

What specific problems with previous DSM are they trying to fix?

- **false sharing:** two machines r/w different vars on same page
    - m1 writes x, m2 writes y
    - m1 writes x, m2 just reads y

  - **Q:** what does IVY do in this situation?
  - **A:** Ivy will bounce the page between x and y back and forth
- **Write amplification:** a 1-byte write turns into a whole-page transfer

First Goal: eliminate write amplification

- don't send whole page, just written bytes

Big idea: write diffs (to fix write amplification)

Example:

```
m1 and m2 both have x's page as readable
m1 writes x
            m2 just reads x
```

- on M1 write fault:
  - tell other hosts to invalidate but *keep hidden copy*
  - M1 makes hidden copy as well
  - M1 marks the page as R/W
- on M2 read fault:
  - M2 asks M1 for recent modifications
  - M1 "diffs" current page against hidden copy
  - M1 send diffs to M2
  - M2 applies diffs to its hidden copy
  - M2 marks the page as read-only
  - M1 marks the page as read-only

Q: Do write diffs provide sequential consistency?

- At most one writeable copy, so writes are ordered
- No writing while any copy is readable, so no stale reads
- Readable copies are up to date, so no stale reads
- Still sequentially consistent

Q: Do write diffs help with false sharing?
A: No, they help with write amplification

Next goal: allow multiple readers+writers to cope with false sharing

- our solution needs to allow two machines to write the same page
- `=>` don't invalidate others when a machine writes
- `=>` don't demote writers to r/o when another machine reads
- `=>` multiple *different* copies of a page!
  - which should a reader look at?

- diffs help: can merge writes to same page
- but when to send the diffs?
    - no invalidations -> no page faults -> what triggers sending diffs

...so we come to *release consistency*

Big idea: (eager) release consistency (RC)

- *Again:* what should trigger sending diffs?
- Seems like we should be sending the diffs when someone reads the data that was changed. How can we tell someone's reading the data if we won't get a read fault because we did not invalidate other people's pages when it was written by one person?
- no-one should read data w/o holding a lock!
    - so let's assume a lock server
- send out write diffs on release (unlock)
    - to *all* machines with a copy of the written page(s)

Example:

```
lock()
x = 1
unlock() --> diffs all pages, to detect all the writes since
             the last unlock
         --> sends diffs to *all* machines
```

Q: Why detect all writes since the last `unlock()` and not the last `lock()`?

A: See causal consistency discussion below.

Example 1 (RC and false sharing)

```
x and y are on the same page
ax -- acquire lock x
rx -- release lock x

M0: a1 for(...) x++ r1
M1: a2 for(...) y++ r2  a1 print x, y r1
```

What does RC do?

- M0 and M1 both get cached writeable copy of the page
- when they release, each computes diffs against original page
- `M1`'s `a1` causes it to wait until `M0`'s `r1` release
    - so M1 will see M0's writes

Q: What is the performance benefit of RC?

- What does IVY do with Example 1?

    - if `x` and `y` are on the same page, page is bounced back between `M0` and `M1`
- multiple machines can have copies of a page, even when 1 or more writes
    - `=>` no bouncing of pages due to false sharing
    - `=>` read copies can co-exist with writers

Q: Is RC sequentially consistent? No!

- in SC, a read sees the latest write
- M1 won't see M0's writes until M0 releases a lock
    - i.e. M1 can see a stale copy of x, which wasn't allowed under seq const
- so machines can temporarily disagree on memory contents
- if you always lock:
    - locks force order `->` no stale reads `->` like sequential consistency

Q: What if you don't lock?

- Reads can return stale data
- Concurrent writes to same var -> trouble

Q: Does RC make sense without write diffs?

- Probably not: diffs needed to reconcile concurrent writes to same page

Big idea: lazy release consistency (LRC)

- one problem is that when we `unlock()` we update everybody, but not everyone might need the changed data
- only send write diffs to next acquirer of released lock
    - (i.e. when someone calls `lock()` and they need updates to the data)
- lazier than RC in two ways:
    - release does nothing, so maybe defer work to future release
    - sends write diffs just to acquirer, not everyone

Example 2 (lazyness)

```
x and y on same page (otherwise IVY avoids copy too)

M0: a1 x=1 r1
M1:           a2 y=1 r2
M2:                     a1 print x,y r1
```

What does LRC do?

- M2 asks the lock manager who the previous holder of lock 1 was
    - it was M1
- M2 only asks previous holder of lock 1 for write diffs

- M2 does not see M1's modification to `y`, even though on same page
  - because it did not acquire lock 2 using `a2`

What does RC do?

- RC would have broadcast all changes on `x` and `y` to everyone

What does IVY do?

- IVY would invalidate pages and ensure only the writer has a write-only copy
- on reads, the written page is turned to read only and the data is fetched by the readers

Q: What's the performance win from LRC?

- if you don't acquire lock on object, you don't see updates to it
- `=>` if you use just some vars on a page, you don't see writes to others
- `=>` less network traffic

Q: Does LRC provide the same consistency model as RC?

- **No!** LRC hides some writes that RC reveals
- Note: if you use locks correctly, then you should not notice the differences between (E)RC and LRC
- in above example, RC reveals `y=1` to M2, LRC does not reveal
  - because RC broadcasts changes on a lock release
- so `"M2: print x, y"` might print fresh data for RC, stale for LRC
  - depends on whether print is before/after M1's release

Q: Is LRC a win over IVY if each variable on a separate page?

- IVY doesn't move data until the program tries to read it
  - So Ivy is pretty lazy already
- Robert: TreadMarks is only worth it pages are big
- Or a win over IVY plus write diffs?

Do we think all threaded/locking code will work with LRC?

- Do all programs lock every shared variable they read?
- Paper doesn't quite say, but strongly implies "no"!

Example 3 (causal anomaly)

```
M0: a1 x=1 r1
M1:             a1 a2 y=x r2 r1
M2:                              a2 print x, y r2
```

What's the potential problem here?

- Counter-intuitive that M2 might see y=1 but x=0
  - because M2 didn't acquire lock 1, it could not get the changes to `x`

A violation of "causal consistency":

- If write W1 contributed to write W2, everyone sees W1 before W2

Example 4 (there's an argument that this is *natural cod*):

```
M0: x=7    a1 y=&x r1
M1:                     a1 a2 z=y r2 r1
M2:                                       a2 print *z r2
```

In example 4, it's not clear if M2 will learn from M1 the writes that M0 also did and contributed to `y=&x`.

- for instance, if `x` was 1 before it was changed by M0, will M2 see this when it prints `*z`

TreadMarks provides **causal consistency**:

- when you acquire a lock,
- you see all writes by previous holder
- and all writes previous holder saw

How to track what writes contributed to a write?

- Number each machine's releases -- "interval" numbers
- Each machine tracks highest write it has seen from each other machine
  - highest write = the interval # of the last write that machine is aware of
  - a "Vector Timestamp"
- Tag each release with current VT
- On acquire, tell previous holder your VT
  - difference indicates which writes need to be sent
- (annotate previous example)
- when can you throw diffs away?
  - seems like you need to globally know what everyone knows about
  - see "Garbage Collection" section from paper

VTs order writes to same variable by different machines:

```
M0: a1 x=1 r1  a2 y=9 r2
M1:              a1 x=2 r1
M2:                           a1 a2 z = x + y r2 r1

M2 is going to hear "x=1" from M0, and "x=2" from M1.
TODO: what about y?
```

How does M2 know what to do?

Could the VTs for two values of the same variable not be ordered?

```
M0: a1 x=1 r1
M1:              a2 x=2 r2
M2:                           a1 a2 print x r2 r1
```

Summary of programmer rules / system guarantees

1. Each shared variable protected by some lock
2. Lock before writing a shared variable to order writes to same var., otherwise "latest value" not well defined
3. Lock before reading a shared variable to get the latest version
4. If no lock for read, guaranteed to see values that contributed to the variables you did lock

Example of when LRC might work too hard.

```
M0: a2 z=99 r2  a1 x=1 r1
M1:                            a1 y=x r1
```

TreadMarks will send `z` to M1, because it comes before `x=1` in VT order.

- Assuming x and z are on the same page.
- Even if on different pages, M1 must invalidate z's page.
- But M1 doesn't use z.
- How could a system understand that z isn't needed?
  - Require locking of all data you read
  - `=>` Relax the causal part of the LRC model

Q: Could TreadMarks work without using VM page protection?

- it uses VM to
  - detect writes to avoid making hidden copies (for diffs) if not needed
  - detect reads to pages => know whether to fetch a diff
- neither is really crucial
- so TM doesn't depend on VM as much as IVY does
  - IVY used VM faults to decide what data has to be moved, and when
  - TM uses acquire()/release() and diffs for that purpose

Performance?

Figure 3 shows mostly good scaling

- is that the same as "good"?
- though apparently Water does lots of locking / sharing

How close are they to best possible performance?

- maybe Figure 5 implies there is only about 20% fat to be cut

Does LRC beat previous DSM schemes?

- they only compare against their own straw-man eager realease consistency (ERC)
  - not against best known prior work
- Figure 9 suggests not much win, even for Water

Has DSM been successful?

- clusters of cooperating machines are hugely successful
- DSM not so much
  - main justification is transparency for existing threaded code
  - that's not interesting for new apps
  - and transparency makes it hard to get high performance
- MapReduce or message-passing or shared storage more common than DSM

6.824 2015 Lecture 15: Optimism, Causality, Vector Timestamps

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Consistency so far:

- *Concurrency* forces us to to think about meaning of reads/writes
- *Sequential consistency:* everyone sees same read/write order (IVY)
- *Release consistency:* everyone sees writes in unlock order (TreadMarks)

Sequential and release consistency are slow:

- in general, must ask before each operation
- IVY: read faults and write faults -> ask manager
- TreadMarks: acquire and release -> ask lock manager
- Can we get better performance by weakening consistency?

Paxos:

- Also slow; several messages to reach agreement.
  - More than IVY+TreadMarks
- Also, "low" availability
  - If no majority, no progress.
- Not suitable for disconnected operation.

Optimistic Concurrency Control

- Do the operation now (e.g., read/write cached copy)
- Check if it was OK later
- Recover if not OK

A simple example -- optimistic peer-to-peer chat

- We each have a computer attached to internet
- When I type something, send msg. to each participant
- Recv msg -> add to end of chat window

Diagram:

```
m0              m1              m2
\              /\              /\
 \------------/                /
  \                           /
   \-------------------------/
```

Do we care about message ordering for chat?

- Network may deliver in different order at different participants
- Joe: The answer is 40
- Fred: No, it's 41
- Alice: That's correct
- Maybe Sam sees different order:
    - Joe: 40
    - Alice: That's correct

What went wrong in this example?

- Alice "computed" her message based on certain inputs
- Sam can only interpret if he has seen those inputs too

Suppose this is an auction chat program:

```
Joe         Fred        Alice

$10 -->
            20
          <-- -->

                 <-- winner is $20
```

If there were a 4th person, Sam:

```
Joe         Fred        Alice                 Sam

$10 -->                                     sees $10
            20
          <-- -->                           does not see $20

                 <-- winner is $20 -->      sees winner is $20
```

So to Sam this might not make sense. His problem is that Sam didn't know what Alice knew when she sent her message.

Definition: `x` causally precedes `y`

- `x` precedes `y` if:
    - M0 does `x`, then M0 does `y`
    - M0 does `x`, M0 sends msg to M1, M1 does `y`
- transitive closure

- `x` and `y` are generally writes, or msgs, or file versions
- also "`y` causally depends on `x`"

Definition: causal consistency

- if `x` causally precedes `y`, everyone sees `x` before `y`

Pros, cons:

- Pro: no single master
- Con: not a total order on events

Slow implementation of causal consistency

- Unique ID for every msg
- Node keeps set of all msg IDs received -- "history"
- When sending `m`, send current history set, too
- Receiver delays incoming msg `m` until has received everything in `m`'s set

History sets will grow huge -- can we abbreviate?

- Each node numbers its msgs 1, 2, 3, &c
- Deliver each node's msgs in order
- Then history need only include latest # seen from each node
    - H1/4 implies saw 1, 2, 3 also
- This notation doesn't grow over time, unlike history sets
- Called a *Vector Timestamp* or *Version Vector*

Vector Timestamp

- Each node numbers its own actions (sent msgs, in this case)
- VT is a vector of numbers, one slot per node
- Each message sent out with a VT
- `VT[i]=x =>` sender had seen all msgs from node `i` up through `#x`
- the assumption here is that a node broadcasts messages to all other nodes (since we're trying to replicate a system effectively)
- have to know how many nodes there are in the whole system
    - otherwise, complicated
- VTs get very large when you have thousands of machines

VT comparisons

- to answer "should msg A be displayed before msg B?"
- let `a` and `b` denote the VTs associated with msgs `A` and `B`

- we can reason about causality (i.e. is `a < b` or are they concurrent `a || b`)
- four situations: `a < b, a || b`
- `a < b` if two conditions hold:
  1. For all hosts `i`:
     - `a[i] <= b[i]`
       - i.e. `a` summarizes a proper prefix of `b`
       - i.e. either
         - `b`'s sender and `a`'s sender have both seen the same # of messages from host `i`
         - `b`'s sender has seen more recent message from host `i` than `a`'s sender has seen
  2. *AND* there exists `j, s.t. a[j] < b[j]`
     - i.e. `a` causally precedes `b`
       - `b`'s sender has *definitely* seen more recent message from host `i` than `a`'s sender has seen
- `a || b` if:
  - exists i,j: `a[i] < b[i]` and `a[j] > b[j]`
  - i.e. neither summarizes a prefix of the other
  - i.e. neither causally precedes the other
    - this is because, as we said before, there's no total order

Many systems use VT variants, but for somewhat different purposes

- TreadMarks, Ficus, Bayou, Dynamo, &c
- compact way to say *"I've seen everyone's updates up to this point"*
- compact way to agree whether event `x` preceded event `y`
- I am pretending there's one fundamental principle here
  - but it's only true if you stand fairly far back

CBCAST -- "causal broadcast" protocol

- General-purpose ordering protocol, useful for peer-to-peer chat
- From Cornell Isis research project
- Key property:
  - Delivers messages to individual nodes in causal order
  - If `a` causally precedes `b`, CBCAST delivers `a` first

[diagram: node, msg buf, VC, chat app]

```
    APP         ^
        |       |
    -----|------|-----------
       \ /      |  CBCAST
        .
    ---------      vector
    | m3    |      clock
    ---------      VT
    | wait  |
    ---------
    | m1    |
```

- Each node keeps a local vector clock, `VC`
  - `VCi[j] = k` means node `i` has seen all msgs from `j` up through message `k`
  - Summarizes what the application has also seen
- `send(m)` at node `i` :
  - `VCi[i] += 1`
  - `broadcast(m, i, VCi)`
- on `receive(m, i, mv)` at node `j` :
  - `j` 's CBCAST library buffers the message
  - release to application only when:
    - `mv <= VCj` , except `mv[i] = VCj[i] + 1`
    - i.e. node `j` has seen every msg that causally precedes `m` `VCj[i] = mv[i]`
    - so msgs will reflect receipt of `m`

Code:

```
on receive(message m, node i, timestamp v):
    release when:
        this node's vector clock VT >= v EXCEPT FOR v[i] = VT[i] + 1
```

Example:

```
    All VCs start <0,0,0>
    M0 sends msg1 w/ <1,0,0>
    M1 receives msg1 w/ <1,0,0>
    M1 sends msg2 w/ <1,1,0>
    M2 receives msg2 w/ <1,1,0> -- must delay because don't have msg1
    M2 receives msg1 w/ <1,0,0> -- can process, unblocks other msg
```

Why fast?

- No central manager, no global order
- If no causal dependencies, CBCAST doesn't delay messages
- Example:
  - `M0 sends <1,0>`
  - `M1 sends <0,1>`
  - Receivers are allowed to deliver in either order

Causal consistency still allows more surprises than sequential

- Sam can still see:
  - Joe: 40
  - Fred: 41
  - Bob: 42
  - Alice: That's correct
- Did she mean 42 or 41?
- Causal consistency only says Alice's msg will be delivered after
  - all msgs she had seen when she sent it
- *Not* that it will be delivered before all msgs she hadn't seen
  - `=>` if CBCAST present `x` and then `y` that does *not* imply `x` happened before `y` necessarily

TreadMarks uses VTs to order writes to same variable by different machines:

```
M0: a1 x=1 r1    a2 y=9 r2
M1:              a1 x=2 r1
M2:                           a1 a2 z=x+y r2 r1

Could M2 hear x=2 from M1, then x=1 from M0?
How does M2 know what to do?
```

VTs are often used for optimistic updating of replicated data

- Everyone has a copy, anyone can write
- Don't want IVY-style MGR or locking: network delays, failures
- Need to sync replicas, accept only "newest" data, detect conflicts
- File sync (Ficus, Coda, Rumor)
- Distributed DBs (Amazon Dynamo, Voldemort, Riak)

File synchronization -- e.g. Ficus

- Multiple computers have a copy of all files
- Each can modify its local copy
- Merge changes later -- optimistic
- fie synchronization with disconnected operation support
  - two people edit the same file on two different airplanes :)
  - when they get back online, server needs to detect this
  - ...and solve it
  - ...and not lose updates (lazy server can just throw away one set of changes)

Scenario:

- user has files replicated at work, at home, on laptop
- hosts may be off, on airplane, &c -- not always on Internet
- work on `H1` for a while, sync changes to `H2`
- work on `H2`, sync changes to `H3`
- work on `H3`, sync to `H1`
- **Overall goal:** push changes around to keep machines identical

Constraint: No Lost Updates

- Only OK for sync to copy version `x2` over version `x1` if
  - `x2` includes all updates that are in `x1`.

Example 1:

```
Focus on a single file

H1: f=1 \----------\
H2:      \->  f=2   \               /--> ???
H3:                  \-> tell H2 --/

What is the right thing to do?
Is it enough to simply take file with latest modification time?
Yes in this case, as long as you carry them along correctly.
  I.e. H3 remembers mtime assigned by H1, not mtime of sync.
```

Example 2:

```
 mtime = 10 | mtime = 20 | mtime = 25

H1: f=1 --\       f=2              /-->
H2:        \-->             f=0 --/
H3:

H2's mtime will be bigger.

Should the file synchronizer use "0" and discard "2"?
  No! They were conflicting changes. We need to detect this case.
  Modification times are not enough by themselves
```

What if there were concurrent updates?

- So that neither version includes the other's updates?
- Copying would then lose one of the updates
- So sync doesn't copy, declares a "conflict"
- Conflicts are a necessary consequence of optimistic writes

How to decide if one version contains all of another's updates?

- We could record each file's entire modification history.
- List of hostname/localtime pairs.
- And carry history along when synchronizing between hosts.
- For example 1: `H2: H1/T1,H2/T2 H3: H1/T1`

- For example 2: `H1: H1/T1,H1/T2 H2: H1/T1,H2/T3`
- Then its easy to decide if version `x` supersedes version `y` :
  - If `y` 's history is a prefix of `x` 's history.

We can use VTs to compress these histories!

- Each host remembers a VT per file
- Number each host's writes to a file (or assign wall-clock times)
- Just remember # of last write from each host
- `VT[i]=x` => file version includes all of host `i` 's updates through `#x`

VTs for Example 1:

- After H1's change: `v1=<1,0,0>`
- After H2's change: `v2=<1,1,0>`
- `v1 < v2` , so H2 ignores H3's copy (no conflict since `<`)
- `v2 > v1` , so H1/H3 would accept H2's copy (again no conflict)

VTs for Example 2:

- After H1's first change: `v1=<1,0,0>`
- After H1's second change: `v2=<2,0,0>`
- After H2's change: `v3=<1,1,0>`
- v3 neither `<` nor `>` v1
  - thus neither has seen all the other's updates
  - thus there's a conflict

What if there *are* conflicting updates?

- VTs can detect them, but then what?
- Depends on the application.
- *Easy:* mailbox file with distinct immutable messages, just union.
- *Medium:* changes to different lines of a C source file (diff+patch).
- *Hard:* changes to the same line of C source.
- Reconciliation must be done manually for the hard cases.
- Today's paper is all about reconciling conflicts

How to think about VTs for file synchronization?

- They detect whether there was a serial order of versions
- I.e. when I modified the file, had I already seen your modification?
  - If yes, no conflict
  - If no, conflict
- Or:
  - A VT summarizes a file's complete version history

  - There's no conflict if your version is a prefix of my version

What about file deletion?

- Can H1 just forget a file's VT if it deletes the file?
  - No: when H1 syncs w/ H2, it will look like H2 has a new file.
- H1 must remember deleted files' VTs.
- Treat delete like a file modification.
  - `H1: f=1 ->H2`
  - `H2: del ->H1`
  - second sync sees `H1:<1,0> H2<1,1>`, so delete wins at H1
- There can be delete/write conflicts
  - `H1: f=1 ->H2 f=2`
  - `H2: del ->H1`
  - `H1:<2,0> vs H2:<1,1> -- conflict`
  - Is it OK to delete at H1?

How to delete the VTs of deleted files?

Is it enough to wait until all hosts have seen the delete msg?

- Sync would carry, for deleted files, set of hosts who have seen del

"Wait until everyone has seen delete" doesn't work:

- `H1: ->H3 forget`
- `H2: f=1 ->H1,H3 del,seen ->H1 ->H1`
- `H3: seen ->H1`
- `H2 needs to re-tell H1 about f, deletion, and f's VT`
  - H2 doesn't know that H3 has seen the delete
  - So H3 might synchronize with H1 and it *would* then tell H1 of f
  - It would be illegal for to to disappear on H1 and re-appear
- So -- this scheme doesn't allow hosts to forget reliably

Diagram:

```
                 | Phase 1               | Phase 2               | Phase 3 (forget f's
VT)
H1: del f  \     | seen f  -\->          | done f  -\->          |
H2:         \--> | seen f  -/-> (bcast)  | done f  -/-> (bcast)  |
H3:         |--> | seen f  -\->          | done f  -\->          |
```

Working VT GC scheme from Ficus replicated file system

- *Phase 1:* accumulate set of nodes that have seen delete
  - terminates when == complete set of nodes

- *Phase 2:* accumulate set of nodes that have completed Phase 1
    - when == all nodes, can totally forget the file
- If H1 then syncs against H2,
    - H2 must be in Phase 2, or completed Phase 2
    - if in Phase 2, H2 knows H1 once saw the delete, so need not tell H1 abt file
    - if H2 has completed Phase 2, it doesn't know about the file either

A classic problem with VTs:

- Many hosts -> big VTs
- Easy for VT to be bigger than the data!
- No very satisfying solution

Many file synchronizers don't use VTs -- e.g. Unison, rsync

- File modification times enough if only two parties, or star
- Need to remember "modified since last sync"
- VTs needed if you want any-to-any sync with > 2 hosts

Summary

- Replication + optimistic updates for speed, high availability
- Causal consistency yields sane order of optimistic updates (CBCAST)
- Causal ordering detects conflicting updates
- Vector Timestamps compactly summarize update histories

6.824 2015 Lecture 12: Eventual Consistency

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Exam

- Bring papers and lecture notes for exam

Bayou: Eventual consistency

- a set of copies of the data, where applications can use any copy of the data
- local read/write
- even if the network breaks, I can still use the local copy
  - *disconnected operation*
- ad-hoc synchronization
  - laptop, phone, tablet can synchronize amongst each other instead of relying on Internet connection
- can work with database servers that have different data and synchronize with each other
- similar to Ficus, but Bayou has more sophisticated conflict resolution

Conflicts

- what to do about the inevitable conflicts that happen when you allow people to write to their local copies and synchronize them later

Meeting room scheduler

Traditional approach (central server):

```
     PDA
 |-----------------
 |9am    824 staff     |----------------|
 |--                   |  Server        |
 |10am        -------------> | DB   |    |
 |--                         | 9am  |    |
 |11am                       | 10am |    |
 |--                                     |
 |12pm                                   |
 |--
```

Not a good approach because it requires everyone to have connectivity to the server.

Would be nice if you have PDA send appointment to laptop, which can then send it to the server.

```
    PDA
|-----------------
|9am    824 staff     |----------------|
|--                   |  Server        |
|10am                       | DB   |   |
|--                         | 9am  |   | <-----\
|11am                       | 10am |   |        \
|--                                    |         |
|12pm                                  |      laptop
|--      \                                      /
          \---------------------------------->/
```

Update functions

Main idea: Update functions. Instead of the application saying "write this DB record", the application hands a function that behaves differently based on what's in the DB.

Example:

- if free at 10am
    - reserve @10am
- else if free at 9am
    - reserve @9am
- else
    - reserve

Bayou takes this function from the PDA and gives it to the laptop.

Suppose A and B want the same times:

- A wants: either staff meeting at 10 or 11
- B wants: hiring meeting at 10 or 11

If you simply apply these functions to node A's and B's databases, that's not enough:

- X syncs with A: X gets 10am staff meeting
- X syncs with B: X gets 11am hiring meeting
- Y syncs with B: Y gets 10am hiring meeting
- Y syncs with A: Y gets 11am staff meeting
- **Bad:** now X and Y have differing views

`=>` have to execute `A`'s and `B`'s update functions in the same order

Numbering updates

Next idea: number update functions, so that you can view them as being a log

- Classic way to order things is to stamp them with numbers and sorting
- initially let the Bayou update ID be `<time T, nodeId>`
  - possible for time `T` to be the same for two update IDs, but then the node IDs will differ (presumably)
- ordering rules:
  - `a < b` if `a.T < b.T` or `a.T == b.T and a.ID < b.ID`

If we take the previous example:

```
<T=10, nodeId=A>, A wants: either staff meeting at 10  or 11
<T=20, nodeId=B>, B wants: hiring meeting at 10 or 11
```

- When Y syncs with B and then with A, it'll see A's update occurred earlier
- so it undoes B's update, applies A's and then B's again

We need to be able to roll back and re-execute the log.

Are the updates consistent with causality?

- PDA A adds a meeting
- A synchronizes with B
- B deletes A's meeting

If some 3rd node sees these updates, it would be necessary to have the meeting creation timestamp be smaller than the deletion timestamp.

Lamport logical clock

Each node maintains `T_max`, the highest timestamp this node has ever seen from itself or from another node.

When a node creates an event and adds it to the log, it picks timestamp `T = max (T_max + 1, wall clock time)`

- new timestamps are always higher than timestamps the node has ever seen

Tentative entries, commit scheme

It's annoying that entries in the calendar are always displayed as tentative because another (earlier) update could come in and replace it.

- maybe because the new update sender was disconnected for a long time

We're looking for a way to all agree that anything above a certain point in the log will never change (it's frozen, no one can modify stuff there)

Bad idea: One possibility is to have all the replicas exchange summary w/ each other about what they've seen:

- X has seen all A's updates through 20, B's through 17, and C's through 72
  - these are timestamps (logical clocks)
- we know that X will never create a timestamp less than 72
- similarly, node Y also has a min timestamp that he will generate next
  - say 30
- we can take the minimum over all these minimums `min(30, 72) = 30` and commit all operations up to that point
- problem is it requires every node to be up and connected to all other nodes

Commit scheme for Bayou

They have one magic node, a primary. Every update that passes through the primary, the primary stamps it with a *commit sequence number* (CSN), the actual ordering number becomes: `<csn, T, node ID>`

- primary does not wait for earlier updates (with smaller `T`) to arrive first, it just timestamps things as they come
- commit preserves causal order
- commit does not preserve wall clock order

If you don't have a CSN: `<-, T, nodeID>` , all commited operations are considered to occur before uncommitted ones.

TODO: not clear what this example was supposed to show

- A's meeting created
- B's meeting created
- B synchronizes with C
- B synchronizes with A
- C synchronizes with primary
- primary applies CSN to A's op, but not B's
- B synchronizes with primary

Vector timestamps

Synchronization

- A has

  - `<-, 10, X>`
  - `<-, 20, Y>`
  - `<-, 30, X>`
  - `<-, 40, X>`
- B has
  - `<-, 10, X>`
  - `<-, 20, Y>`
  - `<-, 30, X>`
- A syncs with B
  - sends a version vector to B describe which updates it has from every node
    - A: `[X 40, Y 20]`
    - (remember that the timestamps are always increased by senders)
    - B: `[X 30, Y 20]`
    - If B compares A's VT with his, he notices that he needs updates by X between timestamp 30 and 40

A new node joins

Now some VTs will have an entry for some new node Z. For instance, in the previous example

- A can send [X 40, Y 20, Z 60] to B

We also need a way to remove nodes.

But B won't know if `Z` is newly added or newly deleted?

- Z joins the system
- Z talks to X
- X generates Z's unique node ID
  - Z's ID = `<Tz, X's node ID>`, where Tz is the time Z talked to X
- X sends an update timestamped with `<-, Tz, X>` that says "new server z"

  - Everybody will see this first before seeing Z's updates

    - Z's updates have timestamps higher than Tz
  - note that IDs are unbounded in size

Forgetting nodes:

- Z's ID = `<20, X>`
- A syncs -> B
- A has log entry from Z `<-, 25, <20, X>>`
- B has no VT entry for Z

Now B needs to figure out from A's updates if Z was added or removed

Case 1: If B's VT entry for `X` is less than the timestamp in `Z` 's ID, then that means that `B` hasn't even seen the creation for `Z` , let alone any updates from `Z` => `B` should create the entry for `Z` because `Z` is new to `B`

Case 2: If B's VT entry for `X` is higher than the timestamp in `Z` 's ID, (ie. B has seen updates from `X` after it created `Z`), then B must've seen `Z` 's creation `=>` B must have seen a deletion notice

Q: If Z's entry is missing from `B` then `Z` (probably?) says `<-, T, Z> bye, T > Tz`

6.824 notes

Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System
Terry, Theimer, Petersen, Demers, Spreitzer, Hauser, SOSP 95

Some material from Flexible Update Propagation for Weakly Consistent Replication, SOSP 97

Why this paper?

- Eventual consistency is pretty common
  - git, iPhone sync, Dropbox, Amazon Dynamo
- Why do people like eventual consistency?
  - fast read/write of local copy (no primary, no paxos)
  - disconnected operation
- What goes wrong?
  - doesn't look like "single copy" (no primary, no paxos)
  - conflicting writes to different copies
  - how to reconcile them when discovered?
- Bayou has the most sophisticated reconciliation story

Paper context:

- Early 1990s (like Ficus)
- Dawn of PDAs, laptops, tablets
  - H/W clunky but clear potential
  - Commercial devices did not have wireless
- Devices might be off or not have network access
  - This problem has not gone away!

   - iPhone sync, Dropbox sync, Dynamo

Let's build a conference room scheduler

- Only one meeting allowed at a time (one room).
- Each entry has a time and a description.
- We want everyone to end up seeing the same set of entries.

Traditional approach: one server

- Server executes one client request at a time
- Checks for conflicting time, says yes or no
- Updates DB
- Proceeds to next request
- Server implicitly chooses order for concurrent requests

Why aren't we satisfied with central server?

- I want to use scheduler on disconnected iPhone &c
   - So need DB replica in each node.
   - Modify on any node, as well as read.
- Periodic connectivity to net.
- Periodic direct contact with other calendar users (e.g. bluetooth).

Straw man 1: merge DBs

- Similar to iPhone calendar sync, or file sync.
- May need to compare every DB entry -- lots of time and net b/w.
- Still need a story for conflicting entries, i.e. two meetings at same time.
   - User may not be available to decide at time of DB merge.
   - So need automatic reconciliation.

Idea for conflicts: update functions

- Application supplies a function, not a new value.
- Read current state of DB, decide best change.
- E.g. "Meet at 9 if room is free at 9, else 10, else 11."
   - Rather than just "Meet at 9"
- Function can make reconciliation decision for absent user.
- Sync exchanges functions, not DB content.

Problem: can't just apply update functions to DB replica

- A's fn: staff meeting at 10:00 or 11:00
- B's fn: hiring meeting at 10:00 or 11:00

- X syncs w/ A, then B
- Y syncs w/ B, then A
- Will X put A's meeting at 10:00, and Y put A's at 11:00?

Goal: eventual consistency

- OK for X and Y to disagree initially
- But after enough syncing, all nodes' DBs should be identical

Idea: ordered update log

- Ordered log of updates at each node.
- Syncing == ensure both nodes have same updates in log.
- DB is result of applying update functions in order.
- Same log `=>` same order `=>` same DB content.

How can nodes agree on update order?

- Update ID: `<time T, node ID>`
- T is creating node's wall-clock time.
- Ordering updates a and b:
    - `a < b` if `a.T < b.T` or (`a.T = b.T` and `a.ID < b.ID`)

Example:

```
<10,A>: staff meeting at 10:00 or 11:00
<20,B>: hiring meeting at 10:00 or 11:00

what's the correct eventual outcome?
  the result of executing update functions in timestamp order
  staff at 10:00, hiring at 11:00
```

What DB content before sync?

- A: staff at 10:00
- B: hiring at 10:00
- This is what A/B user will see before syncing.

Now A and B sync with each other

- Each sorts new entries into its log, order by time-stamp
- Both now know the full set of updates
- A can just run B's update function
- But B has *already* run B's operation, too soon!

Roll back and replay

- B needs to to "roll back" DB, re-run both ops in the right order

- Big point: the log holds the truth; the DB is just an optimization
- We will optimize roll-back in a bit

Displayed meeting room calendar entries are "tentative"

- B's user saw hiring at 10, then it changed to hiring at 11

Will update order be consistent with wall-clock time?

- Maybe A went first (in wall-clock time) with `<10,A>`
- Node clocks unlikely to be perfectly synchronized
- So B could then generate <9,b>
- B's meeting gets priority, even though A asked first
- Not "externally consistent"

Will update order be consistent with causality?

- What if A adds a meeting,
  - then B sees A's meeting,
  - then B deletes A's meeting.
- Perhaps
  - `<10,A> add`
  - `<9,B> delete` -- B's clock is slow
- Now delete will be ordered before add!

Lamport logical clocks for causal consistency

- Want to timestamp events s.t.
  - if node observes E1, then generates E2, then `TS(E2) > TS(E1)`
- So all nodes will order E1, then E2
- `Tmax` = highest time-stamp seen from any node (including self)
- `T = max(Tmax + 1, wall-clock time)` -- to generate a timestamp
- Note properties:
  - E1 then E2 on same node `=> TS(E1) < TS(E2)`
  - BUT
  - `TS(E1) < TS(E2)` does not imply E1 came before E2

Logical clock solves add/delete causality example

- When B sees `<10,A>`,
  - B will set its Tmax to 10, so
  - B will generate `<11,B>` for its delete

Irritating that there could always be a long-delayed update with lower TS

- That can cause the results of my update to change
    - User can never be sure if meeting time is final!
- Would be nice if updates were eventually "stable"
    - `=>` no changes in update order up to that point
    - `=>` results can never again change -- you know for sure when your meeting is
    - `=>` don't have to roll back, re-run committed updates

Bad idea: a fully decentralized "commit" scheme

- Proposal: `<10,A>` is stable if all nodes have seen all updates w/ `TS <= 10`
- Have sync always send in log order -- "prefix property"
- If you have seen updates w/ `TS > 10` from *every* node
    - Then you'll never again see one `< <10,A>`
    - So `<10,A>` is stable
- Why doesn't Bayou do this?
    - Not all nodes are connected to each other

How does Bayou commit updates, so that they are stable?

- One node designated "primary replica".
- It marks each update it receives with a permanent CSN.
    - Commit Sequence Number.
    - That update is committed.
    - So a complete time stamp is `<CSN, local-time, node-id>`
    - Uncommitted updates (are considered to) come after all committed updates w/ this new timestamping scheme
- CSN notifications are synced between nodes.
- The CSNs define a total order for committed updates.
    - All nodes will eventually agree on it.

Will commit order match tentative order?

- Often.
- Syncs send in log order (prefix property)
    - Including updates learned from other nodes.
- So if A's update log says
    - `<-,10,X>`
    - `<-,20,A>`
- A will send both to primary, in that order
    - Primary will assign CSNs in that order
    - Commit order will, in this case, match tentative order

Will commit order always match tentative order?

- No: primary may see newer updates before older ones.
- A has just: `<-,10,A> W1`
- B has just: `<-,20,B> W2`
- If `C` sees both, C's order: `W1 W2`
- B syncs with primary, gets `CSN=5`.
- Later A syncs w/ primary, gets `CSN=6`.
- When C syncs w/ primary, its order will change to `W2 W1`
    - `<5,20,B> W1`
    - `<6,10,A> W2`
- So: committing may change order.

Committing allows app to tell users which calendar entries are stable.

- A stable meeting room time is final.

Nodes can discard committed updates.

- Instead, keep a copy of the DB as of the highest known CSN
- Roll back to that DB when replaying tentative update log
- Never need to roll back farther
    - Prefix property guarantees seen `CSN=x => seen CSN<x`
    - No changes to update order among committed updates

How do I sync if I've discarded part of my log?

- Suppose I've discarded all updates with CSNs.
- I keep a copy of the stable DB reflecting just discarded entries.
- When I propagate to node `X`:
    - If node X's highest CSN is less than mine,
        - I can send him my stable DB reflecting just committed updates.
        - Node X can use my DB as starting point.
        - And X can discard all CSN log entries.
        - Then play his tentative updates into that DB.
    - If node X's highest CSN is greater than mine,
        - X doesn't need my DB.
- In practice, Bayou nodes keep the last few committed updates.
    - To reduce chance of having to send whole DB during sync.

How to sync?

- A sending to B
- Need a quick way for B to tell A what to send
- Committed updates easy: B sends its CSN to A
- What about tentative updates?

- A has: `<-,10,X>` `<-,20,Y>` `<-,30,X>` `<-,40,X>`
- B has: `<-,10,X>` `<-,20,Y>` `<-,30,X>`
- At start of sync, B tells A "X 30, Y 20"
  - Sync prefix property means B has all X updates before 30, all Y before 20
- A sends all X's updates after `<-,30,X>`, all Y's updates after `<-,20,X>`, &c
- This is a version vector -- it summarize log content
  - It's the "F" vector in Figure 4
  - A's F: `[X:40,Y:20]`
  - B's F: `[X:30,Y:20]`

How could we cope with a new server Z joining the system?

- Could it just start generating writes, e.g. `<-,1,Z>` ?
- And other nodes just start including Z in VVs?
- If A syncs to B, A has `<-,10,Z>`, but B has no Z in VV
  - A should pretend B's VV was `[Z:0,...]`

What happens when Z retires (leaves the system)?

- We want to stop including Z in VVs!
- How to announce that Z is gone?
  - Z sends update `<-,?,Z> "retiring"`
- If you see a retirement update, omit Z from VV
- How to deal with a VV that's missing Z?
- If A has log entries from Z, but B's VV has no Z entry:
  - e.g. A has `<-,25,Z>`, B's VV is just `[A:20, B:21]`
  - Maybe Z has retired, B knows, A does not
  - Maybe Z is new, A knows, B does not
- Need a way to disambiguate: Z missing from VV b/c new, or b/c retired?

Bayou's retirement plan

- Z joins by contacting some server `X`
- Z's ID is generated by X as `<Tz,X>`
  - Tz is X's logical clock as of when Z joined
  - Note: unbounded ID size
- X issues `<-,Tz,X> "new server Z"`

How does `ID=<Tz,X>` scheme help disambiguate new vs forgotten?

- Suppose Z's ID is `<20,X>`
- A syncs to B
  - A has log entry from `Z <-,25,<20,X>>`
  - B's VV has no Z entry

- One case:
  - B's VV: `[X:10, ...]`
  - `10 < 20` implies B hasn't yet seen X's "new server Z" update
- The other case:
  - B's VV: `[X:30, ...]`
  - `20 < 30` implies B once knew about Z, but then saw a retirement update

Let's step back.

Is eventual consistency a useful idea?

- Yes: people want fast writes to local copies
- iPhone sync, Dropbox, Dynamo, Riak, Cassandra, &c

Are update conflicts a real problem?

- Yes -- all systems have some more or less awkward solution

Is Bayou's complexity warranted?

- I.e. log of update functions, version vectors, tentative operations
- Only critical if you want peer-to-peer sync
  - I.e. both disconnected operation AND ad-hoc connectivity
- Only tolerable if humans are main consumers of data
- Otherwise you can sync through a central server (iPhone, Dropbox)
- Or read locally but send updates through a master (PNUTS, Spanner)

But there's are good ideas for us to learn from Bayou

- Update functions for automatic application-driven conflict resolution
- Ordered update log is the real truth, not the DB
- Logical clock for causal consistency

6.824 2015 Lecture 13: MapReduce

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Intro

- 2nd trip to this paper, talk more about fault tolerance
  - See first lecture
- a real triumph of simplicity for the programmer
- clever design tricks to get good performance

Example: Building an inverted index

- you need an inverted index for a search index
- maps keywords to documents they are found in

Example:

```
doc 31: I am Alex
doc 32: Alex at 8 am
```

The output that we want, is an index: for each word in the input, we want a list of every place that word occurred (document + offset):

```
alex: 31/2, 32/0 ...
am:   31/1, 32/3 ...
```

The actual map/reduce functions for building an inverted index:

```
map(doc file)
    split doc into words
    for each word
        emit(word, {doc #, offset})

reduce(word string, occurrences list<doc #, offset>)
    emit(word, sorted list of ocurrences by doc # and then by offset)
```

Input files

In MapReduce, the input is stored in GFS (Google's file system)

```
Input, M splits, one    R reduce tasks
map function for each
split

---------               ---------------------------------
|       |               |   |   | * |           |   |   |
---------               ----------|----------------------
|       |               |   |   | * |           |   |   |
---------               ----------|----------------------
|       |               |   |   | * |           |   |   |
---------               ----------|----------------------
|       |               .   .   . |
---------               .   .   .  \
|       |               .   .   .   \-----> data for a single reduce task
---------               .   .   .           is all the data in the column
|       |               .   .   .
---------               .   .   .
|       |               .   .   .
---------               .   .   .
```

What happens if the column of data for a reduce worker not fitting in memory? Seems like it would go to disk.

Note that a single reduce call happens for every unique keyword. So, in our inverted index example, this would mean a single reduce call for the keyword "the" which would appear probably a billion times in a large collection of documents. Thus, this will take a while. MapReduce cannot parallelize the work in a reduce call (lost opportunity for certain reduce functions that are composable, like f(reduce(k, l1), reduce(k, l2)) = reduce(k, l1+l2)).

- I think *combiner functions* mentioned in the paper, can alleviate this issue

Performance

- it's all about data movement
  - pushing terrabytes of data across a cluster
  - 1000 machines
    - can maybe push data to RAM (1GB/s) at 1000 GB/s
    - can maybe push data to disk (100MB/s) at 100 GB/s
    - network can run at 1Gbit/s = 100MB/s on a machine
      - for 1000 machine, the wiring is expensive and costs you speed
      - network is usually a tree with servers at the leaves and bigger switches in the internal nodes
      - bottleneck is root switch, which runs at 18GB/s at Google
    - thus, network can only push data at 18GB/s `=>` *bottleneck*

Design insights

Need to cope with the network problem.

Distributed Shared Memory (DSM) is very flexible in that any machine can write memory on any location in (distributed) memory. The problem is that you end up w/ very bandwidth inefficient and latency sensitive systems. If you allow arbitrary reads/writes to data you end up with a bunch of latency-sensitive small data movements across the network.

DSM makes fault tolerance quite difficult, when a single machine dies, because each machine can do whatever it wants (read or write any mem. loc.), so it's hard to checkpoint the system.

Key ideas:

- `Map()` and `Reduce()` work on local data only.
- `Map()` and `Reduce()` only operate on big pieces of data
    - to amortize network cost of sending
- very little interaction between parts of the system
    - maps cannot talk to each other
    - reduces cannot talk to each other
    - maps and reduces cannot talk to each other
        - other than the implicit communication of sending the mapped data to the reduce functions
- give programmer abstract control over the network communication
    - some control over how keys are mapped into the reduce partitions

Input is typically stored striped (64MB chunks) in GFS, over a lot of disks and machines.

- gotta be clever, because this would imply that Map tasks are limited by network bandwidth

MapReduce takes advantage of GFS knowledge, to actually run the map tasks locally on the GFS machines where the file chunks are stored. `=>` increase bandwitdh to maps from 18GB/s to 100GB/s

Intermediate map files generated by map are also stored locally. Downside is that there's a single copy of the data on that one machine and the reduce worker has to talk to it only `=>` limited bandwidth.

- if the machine stops or crashes, the data is lost, have to restart map

Data in GFS is actually replicated (2 or 3 copies), and this gives MapReduce a choice of 2-3 servers that it can run every map task on.

- good for load/balancing (MR master can move slow map tasks to other machines)
    - don't get this benefit for reduce tasks

Output of reduce is stored in GFS `=>` reduce output is written across the network. `=>` total output of MapReduce system is 18GB/s, if that's your cross-section bandwidth.

QOTD

How soon can reduce start after map emitted some data?

Morris: As soon as a column is filled with data `<=>` as soon as all the maps are finished.

Apparently, you could do it as soon as a map task emits a keyword, by feeding values as they are generated in the reduce task's iterator, but performance can be tricky to achieve in that case.

Does MapReduce scale well?

One of the big benefit of a distributed system, is that you *might* be able to speed it up by just buying more machines. Cheaper to buy machines than to pay programmers.

`nx` hardware => `nx` performance?, `n > 1`

As we grow # of machines (10 fold), and input size stays constant `=>` input size has to be decreased (10 fold). Smaller splits (10x smaller).

If we have millions of machines, the splits can be kilobytes in size `=>` network latency will kill our performance.

You can't have more reduce workers than you have keys.

Scalability is limited by

- map split size
- number of keys `>=` # of reduce workers
- network bandwidth (need to buy more "network" too, as we buy more machines)
  - a really important problem

The answer: certainly get some scaling, but not infinite (limited by network)

Fault tolerance

Challenge: if you run big jobs on 1000s of computers, you are sure to get some failures. So cannot simply restart whole computation. Must just redo failed machine's work.

Difficult to achieve for DSM, easier for MapReduce.

Assuming independent failures (also because maps/reduces are independent)

If worker failed:

- can just restart
- can save intermediate output and resume after failure

If maps fail, we have to rerun it, because it stores its output on the same machine, which is done. Master knows what the map was working on, so it can just restart.

If a reduce worker crashes, because they store their output on GFS, on replicated different servers. We have a good chance of not having to recompute, if the reduce worker finished.

Paper's performance eval

Figure 2 in paper. Why does the bandwidth take 60 seconds to achieve 30GB/s?

The MR job has 1800 mappers, and some *poor* master that has to give work to each one. So maybe the master takes a while to contact everyone.

Why only 30GB/s? These are map tasks so no network overhead. Maybe the CPU is the limit? Unlikely. Seems like this is a disk bandwidth issue. 30GB/s / 1800 machines `=>` 17MB/s per disk

Figure 3 in paper. 800 secs for sorting 1TB of data `=>` 1.25GB/s sort throughput

One thing to notice is that the terrabyte of data fits in the memory of the 1800 machines.

On a single machine with enough memory, Morris extrapolated that it would take around 30,000 seconds to sort 1TB of data (takes 2500secs to sort 100GB)

Middle graph says they are only able to move data across the network at 5GB/s. Simply moving 1TB of data will take 200 seconds. And MapReduces moves it more than once: from maps to reduce, from reduce to GFS (multiple times for replication)

Important insight: Computation involves moving data. Not just CPU cycles.

6.824 original notes

```
Why MapReduce?
  Second look for fault tolerance and performance
  Starting point in current enthusiasm for big cluster computing
  A triumph of simplicity for programmer
  Bulk orientation well matched to cluster with slow network
  Very influential, inspired many successors (Hadoop, Spark, &c)

Cluster computing for Big Data
  1000 computers + disks
```

```
    a LAN
    split up data+computation among machines
    communicate as needed
    similar to DSM vision but much bigger, no desire for compatibilty

  Example: inverted index
    e.g. index terabytes of web pages for a search engine
    Input:
      A collection of documents, e.g. crawled copy of entire web
      doc 31: i am alex
      doc 32: alex at 8 am
    Output:
      alex: 31/3 32/1 ...
      am: 31/2 32/4 ...
    Map(document file i):
      split into words
      for each offset j
        emit key=word[j] value=i/j
    Reduce(word, list of d/o)
      emit word, sorted list of d/o

  Diagram:
    * input partitioned into M splits on GFS: A, B, C, ...
    * Maps read local split, produce R local intermediate files (A0, A1 .. AR)
    * Reduce # = hash(key) % R
    * Reduce task i fetches Ai, Bi, Ci -- from every Map worker
    * Sort the fetched files to bring same key together
    * Call Reduce function on each key's values
    * Write output to GFS
    * Master controls all:
      Map task list
      Reduce task list
      Location of intermediate data (which Map worker ran which Map task)

  Notice:
    Input is huge -- terabytes
    Info from all parts of input contributes to each output index entry
      So terabytes must be communicated between machines
    Output is huge -- terabytes

  The main challenge: communication bottleneck
    Three kinds of data movement needed:
      Read huge input
      Move huge intermediate data
      Store huge output
    How fast can one move data?
      RAM: 1000*1 GB/sec =    1000 GB/sec
      disk: 1000*0.1 GB/sec =  100 GB/sec
      net cross-section:        10 GB/sec
    Explain host link b/w vs net cross-section b/w

  How to cope with communication bottleneck
    Locality: split storage and computation the same way, onto same machines
      Because disk and RAM are faster than the network
    Batching: move megabytes at a time, not e.g. little key/value puts/gets
      Because network latency is a worse problem than network throughput
    Programming: let the developer indicate how data should move between machines
      Because the most powerful solutions lie with application structure

  The big programming idea in MapReduce is the key-driven shuffle
    Map function implicitly specifies what and where data is moved -- with keys
    Programmer can control movement, but isn't burdened with details
    Programs are pretty constrained to help with communication:
      Map can only read the local split of input data, for locality and simplicity
      Just one batch shuffle per computation
      Reduce can only look at one key, for locality and simplicity

  Where does MapReduce input come from?
    Input is striped+replicated over GFS in 64 MB chunks
    But in fact Map always reads from a local disk
      They run the Maps on the GFS server that holds the data
    Tradeoff:
```

```
        Good: Map can read at disk speed, much faster than reading over net from GFS s
erver
        Bad: only two or three choices of where a given Map can run
             potential problem for load balance, stragglers

    Where does MapReduce store intermediate data?
      On the local disk of the Map server (not in GFS)
      Tradeoff:
        Good: local disk write is faster than writing over network to GFS server
        Bad: only one copy, potential problem for fault-tolerance and load-balance

    Where does MapReduce store output?
      In GFS, replicated, separate file per Reduce task
      So output requires network communication -- slow
      The reason: output can then be used as input for subsequent MapReduce

    The Question: How soon after it receives the first file of
      intermediate data can a reduce worker start calling the application's
      Reduce function?

    Why does MapReduce postpone choice of which worker runs a Reduce?
      After all, might run faster if Map output directly streamed to reduce worker
      Dynamic load balance!
      If fixed in advance, one machine 2x slower -> 2x delay for whole computation
        and maybe the rest of the cluster idle/wasted half the time

    Will MR scale?
      Will buying 2x machines yield 1/2 the run-time, indefinitely?
      Map calls probably scale
        2x machines -> each Map's input 1/2 as big -> done in 1/2 the time
        but: input may not be infinitely partitionable
        but: tiny input and intermediate files have high overhead
      Reduce calls probably scale
        2x machines -> each handles 1/2 as many keys -> done in 1/2 the time
        but: can't have more workers than keys
        but: limited if some keys have more values than others
          e.g. "the" has vast number of values for inverted index
          so 2x machines -> no faster, since limited by key w/ most values
      Network may limit scaling, if large intermediate data
        Must spend money on faster core switches as well as more machines
        Not easy -- a hot R+D area now
      Stragglers are a problem, if one machine is slow, or load imbalance
        Can't solve imbalance w/ more machines
      Start-up time is about a minute!!!
        Can't reduce that no matter how many machines you buy (probably makes it worse
)
      More machines -> more failures

    Now let's talk about fault tolerance
      The challenge: paper says one server failure per job!
      Too frequent for whole-job restart to be attractive

    The main idea: Map and Reduce are deterministic and functional,
        so MapReduce can deal with failures by re-executing
      Often a choice:
        Re-execute big tasks, or
        Save output, replicate, use small tasks
      Best tradeoff depends on frequency of failures and expense of communication

    What if a worker fails while running Map?
      Can we restart just that Map on another machine?
        Yes: GFS keeps copy of each input split on 3 machines
      Master knows, tells Reduce workers where to find intermediate files

    If a Map finishes, then that worker fails, do we need to re-run that Map?
      Intermediate output now inaccessible on worker's local disk.
      Thus need to re-run Map elsewhere *unless* all Reduce workers have
        already fetched that Map's output.

    What if Map had started to produce output, then crashed:
      Will some Reduces see Map's output twice?
      And thus produce e.g. word counts that are too high?
```

```
    What if a worker fails while running Reduce?
      Where can a replacement worker find Reduce input?
      If a Reduce finishes, then worker fails, do we need to re-run?
        No: Reduce output is stored+replicated in GFS.

    Load balance
      What if some Map machines are faster than others?
        Or some input splits take longer to process?
      Don't want lots of idle machines and lots of work left to do!
      Solution: many more input splits than machines
      Master hands out more Map tasks as machines finish
      Thus faster machines do bigger share of work
      But there's a constraint:
        Want to run Map task on machine that stores input data
        GFS keeps 3 replicas of each input data split
        So only three efficient choices of where to run each Map task

    Stragglers
      Often one machine is slow at finishing very last task
        h/w or s/w wedged, overloaded with some other work
      Load balance only balances newly assigned tasks
      Solution: always schedule multiple copies of very last tasks!

    How many Map/Reduce tasks vs workers should we have?
      They use M = 10x number of workers, R = 2x.
      More => finer grained load balance.
      More => less redundant work for straggler reduction.
      More => spread tasks of failed worker over more machines, re-execute faster.
      More => overlap Map and shuffle, shuffle and Reduce.
      Less => big intermediate files w/ less overhead.
      M and R also maybe constrained by how data is striped in GFS.
        e.g. 64 MByte GFS chunks means M needs to total data size / 64 MBytes

    Let's look at paper's performance evaluation

    Figure 2 / Section 5.2
      Text search for rare 3-char pattern, just Map, no shuffle or reduce
      One terabyte of input
      1800 machines
      Figure 2 x-axis is time, y-axis is input read rate
      60 seconds of start-up time are *omitted*! (copying program, opening input files
)
      Why does it take so long (60 seconds) to reach the peak rate?
      Why does it go up to 30,000 MB/s? Why not 3,000 or 300,000?
        That's 17 MB/sec per server.
        What limits the peak rate?

    Figure 3(a) / Section 5.3
      sorting a terabyte
      Should we be impressed by 800 seconds?
      Top graph -- Input rate
        Why peak of 10,000 MB/s?
        Why less than Figure 2's 30,000 MB/s? (writes disk)
        Why does read phase last abt 100 seconds?
      Middle graph -- Shuffle rate
        How is shuffle able to start before Map phase finishes? (more map tasks than w
orkers)
        Why does it peak at 5,000 MB/s? (??? net cross-sec b/w abt 18 GB/s)
        Why a gap, then starts again? (runs some Reduce tasks, then fetches more)
        Why is the 2nd bump lower than first? (maybe competing w/ overlapped output wr
ites)
      Lower graph -- Reduce output rate
        How can reduces start before shuffle has finished? (again, shuffle gets all fi
les for some tasks)
        Why is output rate so much lower than input rate? (net rather than disk; write
s twice to GFS)
        Why the gap between apparent end of output and vertical "Done" line? (straggle
rs?)

    What should we buy if we wanted sort to run faster?
      Let's guess how much each resource limits performance.
```

```
      Reading input from disk: 30 GB/sec = 33 seconds (Figure 2)
      Map computation: between zero and 150 seconds (Figure 3(a) top)
      Writing intermediate to disk: ? (maybe 30 Gb/sec = 33 seconds)
      Map->Reduce across net: 5 GB/sec = 200 seconds
      Local sort: 2*100 seconds (gap in Figure 3(a) middle)
      Writing output to GFS twice: 2.5 GB/sec = 400 seconds
      Stragglers: 150 seconds? (Figure 3(a) bottom tail)
      The answer: the network accounts for 600 of 850 seconds

    Is it disappointing that sort harnesses only a small fraction of cluster CPU power
?
      After all, only 200 of 800 seconds were spent sorting.
      If all they did was sort, they should sell CPUs/disks and buy a faster network.

    Modern data centers have relatively faster networks
      e.g. FDS's 5.5 terabits/sec cross-section b/w vs MR paper's 150 gigabits/sec
      while CPUs are only modestly faster than in MR paper
      so today bottleneck might have shifted away from net, towards CPU

    For what applications *doesn't* MapReduce work well?
      Small updates (re-run whole computation?)
      Small unpredictable reads (neither Map nor Reduce can choose input)
      Multiple shuffles (can use multiple MR but not very efficient)
        In general, data-flow graphs with more than two stages
      Iteration (e.g. page-rank)

    MapReduce retrospective
      Single-handedly made big cluster computation popular
        (though coincident with big datacenters, cheap machines, data-oriented compani
es)
      Hadoop is still very popular
      Inspired better successors (Spark, DryadLINQ, &c)
```

6.824 2015 Lecture 14: Spark

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Introduction

- MapReduce benefits:
  - Scales
  - Fault tolerance
  - Strategy dealing with stragglers
    - If a map task is low, MapReduce starts another task on a different machine
    - Tasks don't communicate, so easy to have them run twice in parallel
- MapReduce limitations:
  - very rigid form of computation
  - one map phase, one level of communication between maps and reduce, and the reduce phase
  - what if you wanted to build an inverted index **and** then sort it by the most popular keyword `=>` you would need two MapReduce jobs
  - so, cannot properly deal with multi-stage, iterative nor interactive jobs

Users needed more complicated computations `=>` Spark

There were previous solutions that tackled different types of computations individually. Spark's aim was to provide one solution for a general enough model of computation.

Spark

Hard to do DSM while maintaining scalability and fault tolerance properties.

RDDs: Resilient Distributed Datasets

- a Scala object essentially
- immutable
- partitioned across machines

Example (build an RDD of all the lines in a text file that start with "ERROR"):

```
lines = textFile("log.txt")
errors = lines.Filter(_.startsWith("ERROR"))
errors.persist()
errors.count()
```

RDDs are created by:

- by referring to data in external storage
- by *transforming* other RDDs
  - like the `Filter` call above

Actions kick off a computation on the RDD, like the `count()` call in the example.

The `persist()` call tells Spark to hold the RDD in memory, for fast access.

No work is done until the `count()` action is seen and executed (lazy evaluation)

- you can save work by combining all the filters applied before the action
  - faster to read AND filter than to read the file entirely and then do another filter pass

Fault tolerance

Lineage graphs: dependencies between RDDs

```
lines (file)

    |
   \|   filter(_.startsWith("ERROR"))

errors
```

Machines:

```
file = b1 b2 b3 b4 b5

p1 p2 <-\      p3 p4       p5
-----          -----       -----
| M1|   |      |M2 |       |M3 |
-----   |      -----       -----
b1 b2 --/      b3 b4       b5
```

If you lose an RDD's partition like p4 (because M2 failed), you can rebuild it by tracing through the dependency graph and decide (based on what's already computed) where to start and what to recompute.

How do you know on what machines to recompute? In this example, `b3` and `b4` would be replicated on other machines (maybe on `M1` and `M3`), so that's where you would restart the computation.

Comparison

```
           |     RDDs                         |     DSM
-----------*------------------------------|---------------------------
reads      | any type / granularity         |  fine-grained
writes     | only through transformations   |  fine-grained
           |  coarse grained
faults     | recompute                      |  checkpoint (a la Remus)
stragglers | restart job on diff. machine   |  ? no good strategy ?
```

Spark computation expressivity

Spark is pretty general: a lot of existing parallel computation paradigms, like MapReduce, can be implemented easily on top of it

The reason coarse-grained writes are good enough is because a lot of parallel algorithms simply apply the same op over all data.

Partitioning

Can have a custom partitioning function that says "this RDD has 10 partitions, 1st one is all elements starting with `a`, etc.."

If you use your data set multiple times, grouping it properly so that the data you need sits on the same machine is important.

PageRank example

Example:

```
    the "o"'s are webpages
    the arrows are links (sometimes directed, if not just pick a direction)

    1    2
    o<---o <-\
    |\   |   |
    | \  |   |
    |  \ |   |
    *   **   |
    o--->o --/
    3    4
```

Algorithm:

- Start every page with rank 1
- Everyone splits their rank across their neighbours
  - website 1 will give 0.5 to node 3 and node 4 and receive 0.5 from node 2

- Iterate how many times? Until it converges apparently.

Data:

```
RDD1 'links': (url, links)
 - can compute with a map operation

RDD2 'ranks': (url, rank)
 - links.join(ranks) -> (url, (links, rank))
 -      .flatMap(
                (url, (links, rank)))
                     =>
                links.map( l -> (l.dest, rank/n))
            )
 - TODO: not sure why 'rank/n` or how this transformation works
 - store result in RDD3 'contribs'
 - update ranks with contribs
 - ranks = contribs.reduceByKey( _ + _ )
```

Example of bad allocation, because we'll transfer a lot of data:

```
    the squares are machines (partitions)

            links                           ranks
    -------------------------       ------------------------
    |(a,...)|(d,...)|(c,...)|       |(d,1)  |(e,5)  |(c,3) |
    |(b,...)|       |(e,...)|       |       |(a,1)  |(b,1) |
    -------------------------       -----------------------
        \                                     /
         \------------\  /-------------------/
                       \/
                    -------------------------
                    |(a,...)|(d,...)|       |
                    |       |       |       |
                    -------------------------
```

Example with partitioning:

```
            links                           ranks
    -------------------------       ------------------------
    |(a,...)|(c,...)|(e,...)|       |(a,1)  |(c,3)  |(e,3) |
    |(b,...)|(d,...)|       |       |(b,1)  |(d,1)  |      |
    -------------------------       ------------------------


        contribs are easy to compute locally now
```

Does PageRank need communication at all then? Yes, the `contribs` RDD does a `reduceByKey`

TODO: Not sure what it does

Internal representation

RDD methods:

- `partitions` -- returns a list of partitions
- `preferredLocations(p)` -- returns the preferred locations of a partition
  - tells you about machines where computation would be faster
- `dependencies`
  - how you depend on other RDDs
- `iterator(p, parentIters)`
  - ask an RDD to compute one of its partitions
- `partitioner`
  - allows you to specify a partitioning function

6.824 notes

```
Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Com
puting
Zaharia, Chowdhury, Das, Dave, Ma, McCauley, Franklin, Shenker, Stoica
NSDI 2012

Had TreadMarks since 1996, and Distributed Shared Memory is a very general abstraction
. Why use MapReduce? Or why even use TreadMarks?
Say looking through a log, why not implement it using the regular abstractions (socket
s, files etc?)
  Saves a lot of work:
    communication between nodes
    distribute code
    schedule work
    handle failures


The MapReduce paper had a lot of impact on big data analytics: simple and powerful.
But bit too rigid. Other systems proposed fixes:

Dryad (Microsoft 2007): any directed acyclic graph, edges are communication channels,
can be through disk or via TCP.
  + can implement multiple iterations
  + can pipeline through RAM, don't have to go to disk
  - very low level:
      doesn't deal with partitioning of data, want 100,000 mappers? add 100,000 nodes
      what happens if you run out of RAM? (brief mention of "downgrading" a TCP channe
l to a disk file)
  - doesn't checkpoint/replicate, in the middle of the run (so failures can be expensi
ve)

* Pig latin (Yahoo 2008): programming language that compiles to MapReduce. Adds "Datab
ase style" operators, mainly Join
Join: dataset 1 (k1,v1), dataset 2 (k1, v2). ==> (k1, v1, v2), takes cartesian product
 (all tuples of combinations of v1, v2 with same k1)
Example: dataset 1: all clicks on products on website, dataset 2: demographics (age of
 users), want average age of customer per product.
  + allows multiple iterations
  + can express more
  - still has rigidness from MR (writes to disk after map, to replicated storage after
 reduce, RAM)


Spark

A framework for large scale distributed computation.
  An expressive programming model (can express iteration and joins)
  Gives user control over trade off between fault tolerance with performance
     if user frequently perist w/REPLICATE, fast recovery, but slower execution
```

```
     if infrequently, fast execution but slow recovery

Relatively recent release, but used by (partial list) IBM, Groupon, Yahoo, Baidu..
Can get substantial performance gains when dataset (or a major part of it) can fit in
memory, so anticipated to get more traction.
MapReduce is simple

Abstraction of Resilient Distributed Datasets: an RDD is a collection of partitions of
 records.
Two operations on RDDs:
  Transformations: compute a new RDD from existing RDDs (flatMap, reduceByKey)
    this just specifies a plan. runtime is lazy - doesn't have to materialize (compute
), so it doesn't
  Actions: where some effect is requested: result to be stored, get specific value, et
c.
    causes RDDs to materialize.

Logistic regression (from paper):
val points = spark.textFile(...)
        .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
    val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    }.reduce((a,b) => a+b)
    w -= gradient
}

* w is sent with the closure to the nodes
* materializes a new RDD in every loop iteration


PageRank (from paper):
val links = spark.textFile(...).map(...).persist() // (URL, outlinks)
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
     (url, (links, rank)) =>
    links.map(dest => (dest, rank/links.size))
}
// Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
     .mapValues(sum => a/N + (1-a)*sum)
}


What is an RDD (table 3, S4)
  list of partitions
  list of (parent RDD, wide/narrow dependency)
  function to compute
  partitioning scheme
  computation placement hint
Each transformation takes (one or more) RDDs, and outputs the transformed RDD.

Q: Why does an RDD carry metadata on its partitioning?
A: so transformations that depend on multiple RDDs know whether they need to shuffle d
ata (wide dependency) or not (narrow)
Allows users control over locality and reduces shuffles.

Q: Why the distinction between narrow and wide dependencies?
A: In case of failure.
  narrow dependency only depends on a few partitions that need to be recomputed.
  wide dependency might require an entire RDD

Handling faults.
When Spark computes, by default it only generates one copy of the result, doesn't repl
icate. Without replication, no matter if it's put in RAM or disk, if node fails, on pe
rmanent failure, data is gone.
When some partition is lost and needs to be recomputed, the scheduler needs to find a
way to recompute it. (a fault can be detected by using a heartbeat)
```

```
  will need to compute all partitions it depends on, until a partition in RAM/disk, or
 in replicated storage.
  if wide dependency, will need all partitions of that dependency to recompute, if nar
row just one that RDD

So two mechanisms enable recovery from faults: lineage, and policy of what partitions
to persist (either to one node or replicated)
We talked about lineage before (Transformations)

The user can call persist on an RDD.
  With RELIABLE flag, will keep multiple copies (in RAM if possible, disk if RAM is fu
ll)
  With REPLICATE flag, will write to stable storage (HDFS)
  Without flags, will try to keep in RAM (will spill to disk when RAM is full)

Q: Is persist a transformation or an action?
A: neither. It doesn't create a new RDD, and doesn't cause materialization. It's an in
struction to the scheduler.

Q: By calling persist without flags, is it guaranteed that in case of fault that RDD w
ouldn't have to be recomputed?
A: No. There is no replication, so a node holding a partition could fail.
Replication (either in RAM or in stable storage) is necessary

Currently only manual checkpointing via calls to persist.
Q: Why implement checkpointing? (it's expensive)
A: Long lineage could cause large recovery time. Or when there are wide dependencies a
 single failure might require many partition re-computations.

Checkpointing is like buying insurance: pay writing to stable storage so can recover f
aster in case of fault.
Depends on frequency of failure and on cost of slower recovery
An automatic checkpointing will take these into account, together with size of data (h
ow much time it takes to write), and computation time.

So can handle a node failure by recomputing lineage up to partitions that can be read
from RAM/Disk/replicated storage.
Q: Can Spark handle network partitions?
A: Nodes that cannot communicate with scheduler will appear dead. The part of the netw
ork that can be reached from scheduler can continue
  computation, as long as it has enough data to start the lineage from (if all replica
s of a required partition cannot be reached, cluster
  cannot make progress)


What happens when there isn't enough memory?
  - LRU (Least Recently Used) on partitions
    - first on non-persisted
    - then persisted (but they will be available on disk. makes sure user cannot overb
ook RAM)
  - user can have control on order of eviction via "persistence priority"
  - no reason not to discard non-persisted partitions (if they've already been used)

Shouldn't throw away partitions in RAM that are required but hadn't been used.

degrades to "almost" MapReduce behavior
In figure 7, k-means on 100 Hadoop nodes takes 76-80 seconds
In figure 12, k-means on 25 Spark nodes (with no partitions allowed in memory) takes 6
8.8
Difference could be because MapReduce would use replicated storage after reduce, but S
park by default would only spill to local disk, no network latency and I/O load on rep
licas.
no architectural reason why Spark would be slower than MR

Spark assumes it runs on an isolated memory space (multiple schedulers don't share the
 memory pool well).
Can be solved using a "unified memory manager"
Note that when there is reuse of RDDs between jobs, they need to run on the same sched
uler to benefit anyway.
```

```
(from [P09])
Why not just use parallel databases? Commercially available: "Teradata, Aster Data, Ne
tezza, DATAl-
legro (and therefore soon Microsoft SQL Server via Project Madi-
son), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via
the Database Partitioning Feature), and Oracle (via Exadata)"

At the time, Parallel DBMS were
 * Some are expensive and Hard to set up right
 * SQL declarative (vs. procedural)
 * Required schema, indices etc (an advantages in some cases)
 * "Not made here"


Picollo [P10] uses snapshots of a distributed key-value store to handle fault toleranc
e.
- Computation is comprised of control functions and kernel functions.
- Control functions are responsible for setting up tables (also locality), launching k
ernels, synchronization (barriers that wait for all kernels to complete), and starting
 checkpoints
- Kernels use the key value store. There is a function to merge conflicting writes.
- Checkpoints using Chandy-Lamport
* all data has to fit in RAM
* to recover, all nodes need to revert (expensive)
* no way to mitigate stragglers, cannot just re-run a kernel without reverting to a sn
apshot



[P09] "A Comparison of Approaches to Large-Scale Data Analysis", Pavlo et al. SIGMOD'0
9
[P10] Piccolo: Building Fast, Distributed Programs with Partitioned Tables, Power and
Li, OSDI'10
```

6.824 2015 Lecture 15 Spanner

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Intro

Spanner paper, OSDI 2012

- Shattered old assumption: cannot assume that clocks are tightly synchronized
    - tightly synchronized clocks are now feasible in a global scale distributed system: GPS and atomic clocks as independent sources
- *Data model:* immutable versioned data
- built and deployed system in multiple data centers
- Paxos helps you determine order of events. Why do we still need time?
- used synchronized time to allow local reads without locks
- transactions on top of replication
    - two-phase commit across groups of replicas
- concurrency control
    - strict two phase locking with timestamps
- Paxos
    - long-lived leader (timed leases)
    - pipelined (multiple proposals in flight)
    - out-of-order commit, in-order apply

Spanner and 'research'

- team is chock-full of PhDs
- we write research papers when we feel the urge and we have something to say
- cutting edge development, unbelievable scale, but we are not researchers

Historical context

Bigtable paper, OSDI 2006

- started development at end of 2003 (6 PhDs)
- first customer launched on Bigtable mid 2005
- distributed key-value store

    - single-row transactions
    - later added lazy replication
- value proposition
    - scale to large numbers
    - automatic resharding
- Bigtable was one of the progenitors of "NoSQL" or more precisely "of how do you store a lot of data without building a database"
- basic tenets at the time (design assumptions for Bigtable):
    - who needs a database? key-value store suffices
    - who needs SQL? unnecessary for most applications
    - who needs transactions? two-phase commit is too expensive

Why Spanner?

- found that Bigtable is too hard to use
    - users like the power that SQL database give them
    - engineers shouldn't have to code around
        - the lack of transactions
        - the bugs that manifest due to weak semantics provided by lazy replication
    - programmer productivity matters

Megastore, started ca. 2006, built on top of Bigtable

- optimistic concurrency control
- paxos-based replication
    - no long-lived leader (paxos "election" on every write)
    - every paxos message was written to bigtable
- broader class of transactions than bigtable
- SQL-like schema and query languages
- had consistent replication

Dremel, data analysis at Google, started ca. 2008

- column-oriented storage and query engine
- http://research.google.com/pubs/pub36632.html
- popular because it allowed SQL

Transactions

Percolator, general purpose transactions

- snapshot isolation: a normal transaction has one commit point (logically when you

commit, everything happened then)
    - TODO: lookup what this means, because I couldn't write down his explanation
- built on top of Bigtable
- users demanded transactions, but we weren't ready to build that into bigtable

Spanner

- we knew we needed
    - a database
    - SQL
    - consistent replication across data centers
    - general purpose transactions
- the rest was "merely engineering"

TrueTime came along... (story about how they found out about a guy in NY who was working on distributed clocks and they realized it could be useful for their concurrency control)

Globally synchronized clocks

- spanner behaves like a single-machine database
    - consistent replication: replicas all report the same state
    - external consistency: replicas all report the same order of events
- nice semantics

Were we wrong with bigtable

Yes, and no:

- yes for the long-term: didn't know in 2003 what they knew in 2009, didn't have the people or the technology
- no, because lots of people use bigtable at Google

Imagine you are running a startup. What long-term issues can be postponed?

Startup dilemma:

- too much time spent on scalable storage => wasted effort => not done in time => fail
- too little time spent on scalable storage => when they get popular can't scale => fail

What do you have the skill/ability/will/vision to do?

- we could not have built Spanner 10 years ago: or even 5 years ago

- someone told them they should build transactions in, but they didn't do it because they couldn't at the time

Interesting questions

Why has the Bigtable paper had arguably a bigger impact on both the research communities and technology communities?

- research vs. practice

Why do system-researchers insist on building scalable key-value stores (and not databases)?

Lessons

Lesson 0

Timing is everything. Except luck trumps timing.

You can't plan timing when the world is changing: design the best you can for the problems you have in front of you

TrueTime happened due to fortuitous confluence of events and people (i.e. luck). Same with Bigtable. Spanner's initial design (before 2008) was nowhere near what Google has now: they had anti-luck until the project was restarted in 2008.

Lesson 1

Build what you need, and don't overdesign. Don't underdesign either, because you'll pay for it.

Lesson 2

Sometimes ignorance really is bliss. Or maybe luck.

If you have blinders on, you can't overreach. If we had known we needed a distributed replicated database with external consistency in 2004, we would have failed.

Lesson 3

Your userbase matters.

- bigtable was started when Google `< 2000` employees
  - limited # of products
  - not that many engineers
- spanner was started when Google was `10K` employees
  - more products
  - many more engineers, many more junior engineers, many more acquired companies
- productivity of your employees matters

Wrap up

You can't buy luck. You can't plan for luck. But you can't ignore luck.

You can increase your chances to be lucky:

- have strong technical skills
- work on your design sense (find opportunities to learn!)
- build a strong network of colleagues and friends
- learn how to work on a team
- learn what you are good at, and what you are *not* good at
  - be brutally honest with yourself
  - be willing to ask for help
  - admit when you are wrong
  - people don't like working with people that constantly tell them they are wrong

What Spanner lacks?

Maybe disconnected access: Can we build apps that use DBs and can operate offline?

Disconnected operation in Coda file system work.

6.824 notes

Spanner: Google's Globally-Distributed Database, Corbett et al, OSDI 2012

Why this paper?

- modern, high performance, driven by real-world needs
- sophisticated use of paxos
- tackles consistency + performance (will be a big theme)
- Lab 4 a (hugely) simplified version of Spanner

What are the big ideas?

- shard management w/ paxos replication
- high performance despite synchronous WAN replication
- fast reads by **asking only the nearest replica**
- consistency despite sharding (this is the real focus)
- **clever use of time** for consistency
- distributed transactions

This is a dense paper! I've tried to boil down some of the ideas to simpler form.

Sharding

Idea: sharding

- we've seen this before in FDS
- the real problem is managing configuration changes
- Spanner has a more convincing design for this than FDS

Simplified sharding outline (lab 4):

- replica groups, paxos-replicated
    - paxos log in each replica group
- master, paxos-replicated
    - assigns shards to groups
    - numbered configurations
- if master moves a shard, groups eventually see new config
    - `"start handoff Num=7"` op in both groups' paxos logs
    - though perhaps not at the same time
- `dst` can't finish handoff until it has copies of shard data at majority
    - and can't wait long for possibly-dead minority
    - minority must catch up, so perhaps put shard data in paxos log (!)
- `"end handoff Num=7"` op in both groups' logs

Q: What if a Put is concurrent w/ handoff?

- client sees new config, sends Put to new group before handoff starts?
- client has stale view and sends it to old group after handoff?
- arrives at either during handoff?

Q: What if a failure during handoff?

- e.g. old group thinks shard is handed off
    - but new group fails before it thinks so

Q: Can *two* groups think they are serving a shard?

Q: Could old group still serve shard if can't hear master?

Idea: wide-area synchronous replication

- *Goal:* survive single-site disasters
- *Goal:* replica near customers
- *Goal:* don't lose any updates

Considered impractical until a few years ago

- paxos too expensive, so maybe primary/backup?
- if primary waits for ACK from backup
  - 50ms network will limit throughput and cause palpable delay
  - esp if app has to do multiple reads at 50ms each
- if primary does not wait, it will reply to client before durable
- danger of split brain; can't make network reliable

What's changed?

- other site may be only 5 ms away -- San Francisco / Los Angeles
- faster/cheaper WAN
- apps written to tolerate delays
  - may make many slow read requests
  - but issue them in parallel
  - maybe time out quickly and try elsewhere, or redundant gets
- huge # of concurrent clients lets you get hi thruput despite high delay
  - run their requests in parallel
- people appreciate paxos more and have streamlined variants
  - fewer msgs
    - page 9 of paxos paper: 1 round per op w/ leader + bulk preprepare
    - paper's scheme a little more involved b/c they must ensure there's at most one leader
  - read at any replica

Actual performance?

- Table 3
  - pretend just measuring paxos for writes, read at any replica for reads latency
    - why doesn't write latency go up w/ more replicas?
    - why does std dev of latency go down w/ more replicas?
    - r/o a *lot* faster since not a paxos agreement + use closest replica throughput
    - why does read throughput go up w/ # replicas?

    - why doesn't write throughput go up?
    - does write thruput seem to be going down?
  - what can we conclude from Table 3?
    - is the system fast? slow?
  - how fast do your paxoses run?
    - mine takes 10 ms per agreement
    - with purely local communication and no disk
    - Spanner paxos might wait for disk write
- Figure 5
  - `npaxos=5`, all leaders in same zone
  - why does killing a non-leader in each group have no effect? for killing all the leaders ("leader-hard")
    - why flat for a few seconds?
    - what causes it to start going up?
    - why does it take 5 to 10 seconds to recover?
    - why is slope *higher* until it rejoins?

Spanner reads from any paxos replica

- read does *not* involve a paxos agreement
- just reads the data directly from replica's k/v DB
- maybe 100x faster -- same room rather than cross-country

Q: Could we *write* to just one replica?

Q: Is reading from any replica correct?

Example of problem:

- photo sharing site
- i have photos
- i have an ACL (access control list) saying who can see my photos
- i take my mom out of my ACL, then upload new photo
- really it's web front ends doing these client reads/writes

Order of events:

1. W1: I write ACL on group G1 (bare majority), then
2. W2: I add image on G2 (bare majority), then
3. mom reads image -- may get old data from lagging G2 replica
4. mom reads ACL -- may get new data from G1

This system is not acting like a single server!

- there was not really any point at which the image was

- present but the ACL hadn't been updated

This problem is caused by a combination of

- partitioning -- replica groups operate independently
- cutting corners for performance -- read from any replica

How can we fix this?

1. Make reads see latest data
    - e.g. full paxos for reads expensive!
2. Make reads see *consistent* data
    - data as it existed at *some* previous point in time
    - i.e. before #1, between #1 and #2, or after #2
    - this turns out to be much cheaper
    - spanner does this

Here's a super-simplification of spanner's consistency story for r/o clients

- "snapshot" or "lock-free" reads
- assume for now that all the clocks agree
- server (paxos leader) tags each write with the time at which it occurred
- k/v DB stores *multiple* values for each key,
    - each with a different time
- reading client picks a time `t`
    - for each read
        - ask relevant replica to do the read at time `t`
- how does a replica read a key at time `t` ?
    - return the stored value with highest time `<= t`
- but wait, the replica may be behind
    - that is, there may be a write at time `< t`, but replica hasn't seen it
    - so replica must somehow be sure it has seen all writes `<= t`
    - idea: has it seen *any* operation from time `> t` ?
        - if yes, and paxos group always agrees on ops in time order, it's enough to check/wait for an op with time `> t`
        - that is what spanner does on reads (4.1.3)
- what time should a reading client pick?
    - using current time may force lagging replicas to wait
    - so perhaps a little in the past
    - client may miss latest updates
    - but at least it will see consistent snapshot
    - in our example, won't see new image w/o also seeing ACL update

How does that fix our ACL/image example?

1. W1: I write ACL, G1 assigns it time=10, then
2. W2: I add image, G2 assigns it time=15 (> 10 since clocks agree)
3. mom picks a time, for example t=14
4. mom reads ACL t=14 from lagging G1 replica
    - if it hasn't seen paxos agreements up through t=14, it knows to wait so it will return G1
5. mom reads image from G2 at t=14
    - image may have been written on that replica
    - but it will know to *not* return it since image's time is 15
    - other choices of `t` work too.

Q: Is it reasonable to assume that different computers' clocks agree?

- Why might they not agree?

Q: What may go wrong if servers' clocks don't agree?

A performance problem: reading client may pick time in the future, forcing reading replicas to wait to "catch up"

A correctness problem:

- again, the ACL/image example
- G1 and G2 disagree about what time it is

Sequence of events:

1. W1: I write ACL on G1 -- stamped with time=15
2. W2: I add image on G2 -- stamped with time=10

Now a client read at t=14 will see image but not ACL update

Q: Why doesn't spanner just ensure that the clocks are all correct?

- after all, it has all those master GPS / atomic clocks

TrueTime (section 3)

- there is an actual "absolute" time `t_abs`
    - but server clocks are typically off by some unknown amount
    - TrueTime can bound the error
- so `now()` yields an interval: [earliest,latest]
    - earliest and latest are ordinary scalar times
        - perhaps microseconds since Jan 1 1970
- `t_abs` is highly likely to be between earliest and latest

Q: How does TrueTime choose the interval?

Q: Why are GPS time receivers able to avoid this problem?

- Do they actually avoid it?
- What about the "atomic clocks"?

Spanner assigns each write a scalar time

- might not be the actual absolute time
- but is chosen to ensure consistency

The danger:

- W1 at G1, G1's interval is [20,30]
  - is any time in that interval OK?
- then W2 at G2, G2's interval is [11,21]
  - is any time in that interval OK?
- if they are not careful, might get s1=25 s2=15

So what we want is:

- if W2 starts after W1 finishes, then `s2 > s1`
- simplified *"external consistency invariant"* from 4.1.2
- causes snapshot reads to see data consistent w/ true order of W1, W2

How does spanner assign times to writes?

- (again, this is much simplified, see 4.1.2)
- a write request arrives at paxos leader
- `s` will be the write's time-stamp
- leader sets `s` to `TrueTime now().latest`
  - this is "Start" in 4.1.2
- then leader *delays* until `s < now().earliest`
  - i.e. until `s` is guaranteed to be in the past (compared to absolute time)
  - this is "commit wait" in 4.1.2
- then leader runs paxos to cause the write to happen
- then leader replies to client

Does this work for our example?

- W1 at G1, TrueTime says [20,30]
  - `s1 = 30`
  - commit wait until TrueTime says [31,41]
  - reply to client
- W2 at G2, TrueTime *must* now say `>= [21,31]`

    - (otherwise TrueTime is broken)
    - s2 = 31
    - commit wait until TrueTime says [32,43]
    - reply to client
- it does work for this example:
    - the client observed that W1 finished before S2 started,
    - and indeed `s2 > s1`
    - even though G2's TrueTime clock was slow by the most it could be
    - so if my mom sees S2, she is guaranteed to also see W1

Why the "Start" rule?

- i.e. why choose the time at the end of the TrueTime interval?
- previous writers waited only until their timestamps were barely `< t_abs`
- new writer must choose `s` greater than any completed write
- `t_abs` might be as high as `now().latest`
- so s = now().latest

Why the "Commit Wait" rule?

- ensures that `s < t_abs`
- otherwise write might complete with an s in the future
    - and would let Start rule give too low an `s` to a subsequent write

Q: Why commit *wait*; why not immediately write value with chosen time?

- indirectly forces subsequent write to have high enough s
    - the system has no other way to communicate minimum acceptable next s for writes in different replica groups
- waiting forces writes that some external agent is serializing to have monotonically increasing timestamps
- w/o wait, our example goes back to s1=30 s2=21
- you could imagine explicit schemes to communicate last write's TS to the next write

Q: How long is the commit wait?

This answers today's Question: a large TrueTime uncertainty requires a long commit wait so Spanner authors are interested in accurate low-uncertainty time

Let's step back

- why did we get into all this timestamp stuff?
    - our replicas were 100s or 1000s of miles apart (for locality/fault tol)
    - we wanted fast reads from a local replica (no full paxos)
    - our data was partitioned over many replica groups w/ separate clocks

  - we wanted consistency for reads:
    - if W1 then W2, reads don't see W2 but not W1
  - it's complex but it makes sense as a
  - high-performance evolution of Lab 3 / Lab 4

Why is this timestamp technique interesting?

- we want to enforce order -- things that happened in some order in real time are ordered the same way by the distributed system -- "external consistency"
- the naive approach requires a central agent, or lots of communication
- Spanner does the synchronization implicitly via time
  - time can be a form of communication
  - e.g. we agree in advance to meet for dinner at 6:00pm

There's a lot of additional complexity in the paper

- transactions, two phase commit, two phase locking,
  - schema change, query language, &c
- some of this we'll see more of later
- in particular, the problem of ordering events in a distributed system will come up a lot, soon

6.824 2015 Lecture 16: Memcache at Facebook

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Introduction

Facebook Memcached paper:

- an experience paper, not a research results paper
- you can read it as a triumph paper
- you can read it as a caution paper: what happens when you don't think about scalability
- you can read it as a trade-offs paper

Scaling a webapp

Initial design for any webapp is a single webserver machine with a DB server running on it.

Diagram (single machine: webserver and DB server)

```
Single machine
-------------------
| Web app         |
|                 |
|                 |
|                 |
|       | DB | <-----> |Disk|
------------------
```

- eventually they find out they use 100% of the CPU on this machine
- `top` will tell them the CPU time is going to the web-app

Diagram (multiple webserver machines, single DB machine):

```
Web app -----
Web app      \ -----> |DB| <-> Disk
...
Web app
```

- next problem they will face is the database will be the bottleneck
  - DB CPU is 100% or disk is 100%
- say they decide to buy a bunch of DB servers

Diagram:

```
Web app              |DB1| <-> Disk  (users A-M)
Web app              |DB2| <-> Disk  (users N-T)
...                  |DB3| <-> Disk  ...
Web app              ...
```

- now they gotta figure out how to *shard* the data on the database
- application software now has to know which DB server to talk to
- we can no longer have transactions across the whole DB
- we can no longer have single queries on the entire dataset
  - need to send separate queries to each server
- you can't push this too far because after a while the shards get very small and you get database servers that become hotspots

Next, you notice that most of the operations in the DB are reads (if that's the case. It is at Facebook.)

- it turns out you can build a very simple memory cache that can serve half a million requests per second
- then you can remove 90% of the load on the database

Diagram:

```
Web app -> |MC| --> |DB1| <-> Disk  (users A-M)
Web app    |MC|     |DB2| <-> Disk  (users N-T)
...         ..      |DB3| <-> Disk  ...
Web app             ...
```

- the next bottleneck will be database writes, if you keep growing your service

Observation: you could use DB read-only replicas, instead of your own customized memcache (MC) nodes. Facebook did not do this because they wanted to separate their caching logic from their DB deployment:

> It was the best choice given limited engineering resources and time. Additionally, separating our caching layer from our persistence layer allows us to adjust each layer independently as our workload changes

Facebook's use case

Very crucial: they do not care too much about all their users getting a consistent view of their system.

The only case when the paper cares about freshness and consistency is when webapp clients read their own writes.

Their high level picture:

```
Regions (data centers):
    Master region (writable)
   -----------------
  | Web1 Web2 ...   |
  | MC1  MC2  ...   |
  | DB1  DB2  ... <--- complete copy of all data
   -----------------

    Slave regions (read only)
   -----------------
  | Web1 Web2 ...   |
  | MC1  MC2  ...   |
  | DB1  DB2  ... <--- complete copy of all data
   -----------------
```

The reason for having multiple data centers: parallelism across the globe

- maybe also for backup purposes (paper doesn't detail too much)

Big lessons

Look-aside caching can be tricky

This style of look-aside caching, where the application looks in the cache to see what's there, is extremely easy to add to an existing system

- but there are some nasty consistency problems that appear when the caching layer is oblivious to what happens in the DB

Caching is about throughput not latency

It wasn't about reducing latency for the users. They were using the cache to increase throughput and take the load off the database.

- no way the DB could've handled the load, which is 10x or 100x more than what the DB can access

They can tolerate stale data

They want to be able to read their own writes

You'd think this can be easily fixed in the application. Slightly surprised that this was not fixed by just having the application remembering the writes. Not clear why they solved it differently.

Eventual consistency is good enough

They have enormous fan-out

Each webpage they serve might generate hundreds and hundreds of reads. A little bit surprising. So they have to do a bunch of tricks. Issue the reads in parallel. When a single server does this, it gets a bunch of responses back, and the amount of buffering in the switches and webservers is limited, so if they're not careful they can lose packets and thus performance when retrying.

Performance

- a lot of content about consistency in the paper
- but really they were desperate to get performance which led to doing tricks, which led to consistency problems
- performance comes from being able to serve a lot of `Get` 's in parallel

Really only two strategies:

- partition data
- replicate data
- they use both

Partitioning works if keys are roughly all as popular. Otherwise, certain partitions would be more popular and lead to hotspots. Replication helps with handling demand for popular keys. Also, replication helps with requests from remote places in the world.

You can't simply cache keys in the web app servers, because they would all fill their memories quickly and you would double-store a lot of data.

Specific problem they dealt with

Each cluster has a full set of memcache servers and a full set of web servers. Each web server talks to memcache servers in its own cluster.

Adding a new cluster

Sometimes, they want to add a new cluster, which will obviously have empty memcache servers.

- all webservers in new cluster will always miss on every request and will have to go down and contact the DB, which cannot handle the increased load

- instead of contacting the DB, the new cluster will contact memcache servers from other clusters until the new cluster's cache is warmed up

Q: what benefit do they get from adding new clusters? instead of increasing size of existing cluster?

- one possibility is there are some very popular keys so over-partitioning a cluster won't help with that
- another possibility could be that it's easier to add more memcache servers by adding a new cluster, because of the data movement problem

Memcache server goes down

If a memcache server goes down, requests are redirected to a gutter server. The gutter machines will miss a lot initially, but at least it will be caching the results for the future.

Homework question:

Q: Why aren't gutters invalidated on writes?

On a write, DB typically sends an invalidate to all the MC servers that might have that key. So there's a lot of deletes being sent around to a lot of MC servers. Maybe they don't want to overflow the gutter servers with all the deletes.

Note that gutter keys expire after a certain time, to deal with the fact that the keys never change.

Not clear what happens if gutter servers go down.

Q: Wouldn't it better if the DB server sent the out the new value instead of invalidate.

- what's cached in MC, might not be the DB value, but it might be some function of the DB value, that the DB layer is not aware of
    - think of how a friend list is stored in a DB versus how it would be stored in MC

Leases for thundering herds

One client sends an update to the DB and that gets a popular key invalidated. So now lots of lots of clients generate Get's into MC, but the key was deleted, which would lead to lots of DB queries and then lots of caching of the result.

If memcache receives a get for a key that's not present, it will set a lease on that key and say "you're allowed to go ask the DB for this key, but please finish doing this in 10 seconds." When subsequent Get's come in, they are told "no such key, but another guy is getting it, so

please wait for him instead of querying the DB"

The lease is cancelled after 10s or when the owner sets the key.

Each cluster will generate a separate lease.

6.824 notes

```
Scaling Memcache at Facebook, by Nishtala et al, NSDI 2013

why are we reading this paper?
  it's an experience paper, not about new ideas/techniques
  three ways to read it:
    cautionary tale of problems from not taking consistency seriously
    impressive story of super high capacity from mostly-off-the-shelf s/w
    fundamental struggle between performance and consistency
  we can argue with their design, but not their success

how do web sites scale up with growing load?
  a typical story of evolution over time:
  1. one machine, web server, application, DB
     DB stores on disk, crash recovery, transactions, SQL
     application queries DB, formats, HTML, &c
     but the load grows, your PHP application takes too much CPU time
  2. many web FEs, one shared DB
     an easy change, since web server + app already separate from storage
     FEs are stateless, all sharing (and concurrency control) via DB
     but the load grows; add more FEs; soon single DB server is bottleneck
  3. many web FEs, data sharded over cluster of DBs
     partition data by key over the DBs
       app looks at key (e.g. user), chooses the right DB
     good DB parallelism if no data is super-popular
     painful -- cross-shard transactions and queries probably don't work
       hard to partition too finely
     but DBs are slow, even for reads, why not cache read requests?
  4. many web FEs, many caches for reads, many DBs for writes
     cost-effective b/c read-heavy and memcached 10x faster than a DB
       memcached just an in-memory hash table, very simple
     complex b/c DB and memcacheds can get out of sync
     (next bottleneck will be DB writes -- hard to solve)

the big facebook infrastructure picture
  lots of users, friend lists, status, posts, likes, photos
    fresh/consistent data apparently not critical
    because humans are tolerant?
  high load: billions of operations per second
    that's 10,000x the throughput of one DB server
  multiple data centers (at least west and east coast)
  each data center -- "region":
    "real" data sharded over MySQL DBs
    memcached layer (mc)
    web servers (clients of memcached)
  each data center's DBs contain full replica
  west coast is master, others are slaves via MySQL async log replication

how do FB apps use mc?
  read:
    v = get(k) (computes hash(k) to choose mc server)
    if v is nil {
      v = fetch from DB
      set(k, v)
    }
  write:
    v = new value
    send k,v to DB
```

```
    delete(k)
  application determines relationship of mc to DB
    mc doesn't know anything about DB
  FB uses mc as a "look-aside" cache
    real data is in the DB
    cached value (if any) should be same as DB

what does FB store in mc?
  paper does not say
  maybe userID -> name; userID -> friend list; postID -> text; URL -> likes
  basically copies of data from DB

paper lessons:
  look-aside is much trickier than it looks -- consistency
    paper is trying to integrate mutually-oblivious storage layers
  cache is critical:
    not really about reducing user-visible delay
    mostly about surviving huge load!
    cache misses and failures can create intolerable DB load
  they can tolerate modest staleness: no freshness guarantee
  stale data nevertheless a big headache
    want to avoid unbounded staleness (e.g. missing a delete() entirely)
    want read-your-own-writes
    each performance fix brings a new source of staleness
  huge "fan-out" => parallel fetch, in-cast congestion

let's talk about performance first
  majority of paper is about avoiding stale data
  but staleness only arose from performance design

performance comes from parallel get()s by many mc servers
  driven by parallel processing of HTTP requests by many web servers
  two basic parallel strategies for storage: partition vs replication

will partition or replication yield most mc throughput?
  partition: server i, key k -> mc server hash(k)
  replicate: server i, key k -> mc server hash(i)
  partition is more memory efficient (one copy of each k/v)
  partition works well if no key is very popular
  partition forces each web server to talk to many mc servers (overhead)
  replication works better if a few keys are very popular

performance and regions (Section 5)

Q: what is the point of regions -- multiple complete replicas?
   lower RTT to users (east coast, west coast)
   parallel reads of popular data due to replication
   (note DB replicas help only read performance, no write performance)
   maybe hot replica for main site failure?

Q: why not partition users over regions?
   i.e. why not east-coast users' data in east-coast region, &c
   social net -> not much locality
   very different from e.g. e-mail

Q: why OK performance despite all writes forced to go to the master region?
   writes would need to be sent to all regions anyway -- replicas
   users probably wait for round-trip to update DB in master region
     only 100ms, not so bad
   users do not wait for all effects of writes to finish
     i.e. for all stale cached values to be deleted

performance within a region (Section 4)

multiple mc clusters *within* each region
  cluster == complete set of mc cache servers
    i.e. a replica, at least of cached data

why multiple clusters per region?
  why not add more and more mc servers to a single cluster?
  1. adding mc servers to cluster doesn't help single popular keys
     replicating (one copy per cluster) does help
```

```
    2. more mcs in cluster -> each client req talks to more servers
       and more in-cast congestion at requesting web servers
       client requests fetch 20 to 500 keys! over many mc servers
       MUST request them in parallel (otherwise total latency too large)
       so all replies come back at the same time
       network switches, NIC run out of buffers
    3. hard to build network for single big cluster
       uniform client/server access
       so cross-section b/w must be large -- expensive
       two clusters -> 1/2 the cross-section b/w

  but -- replicating is a waste of RAM for less-popular items
    "regional pool" shared by all clusters
    unpopular objects (no need for many copies)
    decided by *type* of object
    frees RAM to replicate more popular objects

  bringing up new mc cluster was a serious performance problem
    new cluster has 0% hit rate
    if clients use it, will generate big spike in DB load
      if ordinarily 1% miss rate, and (let's say) 2 clusters,
        adding "cold" third cluster will causes misses for 33% of ops.
      i.e. 30x spike in DB load!
    thus the clients of new cluster first get() from existing cluster (4.3)
      and set() into new cluster
      basically lazy copy of existing cluster to new cluster
    better 2x load on existing cluster than 30x load on DB

  important practical networking problems:
    n^2 TCP connections is too much state
      thus UDP for client get()s
    UDP is not reliable or ordered
      thus TCP for client set()s
      and mcrouter to reduce n in n^2
    small request per packet is not efficient (for TCP or UDP)
      per-packet overhead (interrupt &c) is too high
      thus mcrouter batches many requests into each packet

  mc server failure?
    can't have DB servers handle the misses -- too much load
    can't shift load to one other mc server -- too much
    can't re-partition all data -- time consuming
    Gutter -- pool of idle servers, clients only use after mc server fails

  The Question:
    why don't clients send invalidates to Gutter servers?
    my guess: would double delete() traffic
      and send too many delete()s to small gutter pool
      since any key might be in the gutter pool

  thundering herd
    one client updates DB and delete()s a key
    lots of clients get() but miss
      they all fetch from DB
      they all set()
    not good: needless DB load
    mc gives just the first missing client a "lease"
      lease = permission to refresh from DB
      mc tells others "try get() again in a few milliseconds"
    effect: only one client reads the DB and does set()
      others re-try get() later and hopefully hit

  let's talk about consistency now

  the big truth
    hard to get both consistency (== freshness) and performance
    performance for reads = many copies
    many copies = hard to keep them equal

  what is their consistency goal?
    *not* read sees latest write
      since not guaranteed across clusters
```

```
  more like "not more than a few seconds stale"
    i.e. eventual
  *and* writers see their own writes
    read-your-own-writes is a big driving force

first, how are DB replicas kept consistent across regions?
  one region is master
  master DBs distribute log of updates to DBs in slave regions
  slave DBs apply
  slave DBs are complete replicas (not caches)
  DB replication delay can be considerable (many seconds)

how do we feel about the consistency of the DB replication scheme?
  good: eventual consistency, b/c single ordered write stream
  bad: longish replication delay -> stale reads

how do they keep mc content consistent w/ DB content?
  1. DBs send invalidates (delete()s) to all mc servers that might cache
     + Do they wait for ACK? I'm guessing no.
  2. writing client also invalidates mc in local cluster
     for read-your-writes

why did they have consistency problems in mc?
  client code to copy DB to mc wasn't atomic:
    1. writes: DB update ... mc delete()
    2. read miss: DB read ... mc set()
  so *concurrent* clients had races

what were the races and fixes?

Race 1: one client's cached get(k) replaces another client's updated k
  k not in cache
  C1: MC::get(k), misses
  C1: v = read k from DB
    C2: updates k in DB
    C2: and DB calls MC::delete(k) -- k is not cached, so does nothing
  C1: set(k, v)
  now mc has stale data, delete(k) has already happened
  will stay stale indefinitely, until key is next written
  solved with leases -- C1 gets a lease, but C2's delete() invalidates lease,
    so mc ignores C1's set
    key still missing, so next reader will refresh it from DB

Race 2: updating(k) in cold cluster, but getting stale k from warm cluster
  during cold cluster warm-up
  remember clients try get() in warm cluster, copy to cold cluster
  k starts with value v1
  C1: updates k to v2 in DB
  C1: delete(k) -- in cold cluster
  C2: get(k), miss -- in cold cluster
  C2: v1 = get(k) from warm cluster, hits
  C2: set(k, v1) into cold cluster
  now mc has stale v1, but delete() has already happened
    will stay stale indefinitely, until key is next written
  solved with two-second hold-off, just used on cold clusters
    after C1 delete(), cold ignores set()s for two seconds
    by then, delete() will propagate via DB to warm cluster

Race 3: writing to master region, but reading stale from local
  k starts with value v1
  C1: is in a slave region
  C1: updates k=v2 in master DB
  C1: delete(k) -- local region
  C1: get(k), miss
  C1: read local DB  -- sees v1, not v2!
  later, v2 arrives from master DB
  solved by "remote mark"
    C1 delete() marks key "remote"
    get()/miss yields "remote"
      tells C1 to read from *master* region
    "remote" cleared when new data arrives from master region
```

```
Q: aren't all these problems caused by clients copying DB data to mc?
   why not instead have DB send new values to mc, so clients only read mc?
     then there would be no racing client updates &c, just ordered writes
A:
  1. DB doesn't generally know how to compute values for mc
     generally client app code computes them from DB results,
       i.e. mc content is often not simply a literal DB record
  2. would increase read-your-own writes delay
  3. DB doesn't know what's cached, would end up sending lots
     of values for keys that aren't cached

PNUTS does take this alternate approach of master-updates-all-copies

FB/mc lessons for storage system designers?
  cache is vital to throughput survival, not just a latency tweak
  need flexible tools for controlling partition vs replication
  need better ideas for integrating storage layers with consistency
```

6.824 2015 Lecture 17: PNUTS

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

PNUTS

- a solution to the same problem Spanner and memcached solved
- PNUTS is a more-principled designed than the memcache Facebook design
  - "it was actually designed"
- make reads fast
- upside: web applications are able to do fast local reads due to replication
- downside: writes will be slow, because they need to be replicated
- because writes have to be distributed to all the regions, there will be a fundamental delay between when writes happen and when the updates actually propagate
  - `=>` potential for stale reads
- if there's data that could be updated by concurrent clients, there will be a problem with multiple writes
  - need all regions to see our writes in the same order

Diagram:

```
Region R1                         Region R2
---------                         ---------

 W1 Mesage broker                  W1 Message broker
 W2     (replicated)               W2     (replicated)
 W3                                W3
 ..          Tablet controller     ..          Tablet controller
                 (replicated)                      (replicated)

    Router1 Router2 ...               Router1 Router2 ...

    SU1 SU2 SU3 ...                   SU1 SU2 SU3 ...
```

- each region has its own set of webservers
- each region stores all data
- each table in a region is partitioned among storage units (SUs)
- routers know the partitioning
- each SU has a disk

Updates

- each record in PNUTS has its own master region through which all writes have to go
  - different than memcache at facebook, they had a master region for *all* records
  - in PNUTS every record has a different master
  - Note: a record is just a row in a table (and has an extra field that stores its master)
- updating records that are in regions far away from the user will take longer of course
- how does the webserver know where to send the update?
  - contact one of the routers
  - router looks at the key, knows it's stored in say SU3
  - find out from SU3 that a different region `r2` has the master copy
    - doesn't know which SU at `r2` the record is at
  - contact one of the routers in `r2`
  - router tells you the SU to store it at
  - the SU then needs to send out the update to all the other regions
  - the SU sends the update to the message brokers
    - not clear if SU applies the update to its own disk before
  - the message broker writes a copy of the update to the disk because it is *committing* to actually sending the update everywhere
    - important because we don't want a failed server to result in partially propagating the update
  - the MB will send it out to other MBs at other sites
  - somehow the web app needs to find out that the write completes
    - not clear who sends the ACK back
    - seems that the MB replies back to the webserver app as soon as it commits the update to the disk
  - asynchronous writes because, from POV of webapp, write is done when MB has written it to its disk
  - why isn't the MB a bottleneck? It has to write a lot of stuff:
    - different applications have a different message broker
    - MB may be able to do much more batching of the writes
    - maybe also MB writes are also less complex than normal database writes where you have to modify Btrees, maybe go through a file system, etc.
  - because they funnel all the writes through some MB they get some semantics for writes

Write semantics

Write order to single records

```
Name        Where       What
----        -----       ----
Alice       home        asleep
Bob
```

- Alice writes where record which has 3 columns (Name, Where, What)
- Alice's application says `write(what=awake)`
  - write goes through PNUTS
- Alice's application says `write(where=work)`
  - write goes through PNUTS
- useful semantics given by PNUTS
  - other people in different regions might see
    - alice at home asleep
    - alice at home awake
    - alice at work awake
  - other people won't see a view of the record inconsistent with the order of the writes
    - alice at work asleep
  - kind of the main consistency semantics provided by PNUTS
  - a result of sequencing the writes through the MBs
  - paper calls this *per-record timeline consistency*
  - note that their model restricts them to only have transactions on a single record basis

When would you care about stale data?

- after you added something to your shopping cart, you would expect to see it there

Reads vs. staleness

```
    read-any(key) -> fast read, just executes the read on the SU and does
                   not wait for any writes to propagate

    read-critical(key, ver) -> returns the read record where ver(record) >= ver
     - useful for reading your own writes
     - true when you have one webpage in a single tab
     - if you update your shopping cart in one tab, then the other tab
       will not be aware of that version number from the first tab

    read-latest(key) -> will always go to the master copy and read the latest
                         data there
```

Writes, atomic updates

Example: increment a counter in a record

```
    test-and-set-write(ver, key, newvalue) -> always gets sent to the master
        region for the key. look at the version and if it matches provided
        one then update the record with the new value

        // implementing v[k]++
        while true:
            (x, v) = read-latest(k)
            if test-and-set-write(k, v, x+1)
                break
```

Question of the day

Alice comes back from spring break and she:

- removes her mom from her ACL
- posts spring break photos

Can her mom see her photos due to out-of-order writes?

If Alice has all the photos her mom can see in a single record, then no.

```
Alice   |   ACL     | List of photos
-------- ----------- ----------------
            mom         p7, p99
```

Assuming the code her mom is executing reads the full record (ACL + photos) when doing the check, and doesn't first read the ACL, wait a while and then read the photos

Failures

If webapp server fails in the middle of doing a bunch of writes, then only partial info would have been written to PNUTS, possibly leading to corruption.

- no transactions for multiple writes

If SU crashes and reboots, it can recover from disk and MB can keep retrying it

What happens when SU loses its disk? It needs to recover the data.

- the paper says the SU will clone its data from a SU from another region
  - main challenge is that updates are being sent by MBs to records that are being copied
  - either updates go to source of copy, or destination of copy remembers the update
  - ultimately they both need to have the update in the end

Performance

Evaluation mostly focuses on latency and not on throughput. Maybe this is specific to their needs.

Not clear how they can support millions of users with MBs that can only do hundreds of writes per second.

Why is it taking them 75ms to do a local update, where everyone is in the same region?

- computation, disk, network?
- 75ms is enormous for a write in a DB

6.824 notes

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. Proceedings of VLDB, 2008.

Why this paper?

- same basic goals as Facebook/memcache paper, more principled design
- multi-region is very challenging -- 100ms network delays
- conscious trade-off between consistency and performance

What is PNUTS' overall goal?

Diagram:

```
[world, browsers, data centers]
```

- overall story similar to that of Spanner and Facebook/memcache
- data centers ("regions") all over the world
- web applications, e.g. mail, shopping, social net
    - each app probably runs at all regions
- PNUTS keeps state for apps
    - per-user: profile, shopping cart, friend list
    - per-item: book popularity, user comments
- app might need any piece of data at any data center
- need to handle lots of concurrent updates to different data
    - e.g. lots of users must be able to add items to shopping cart at same time thus 1000s of PNUTS servers
- 1000s of servers => crashes must be frequent

Overview

Diagram:

```
3 regions, browsers, web apps, tablet ctlrs, routers, storage units, MBs]
```

- each region has all data
- each table partitioned by key over storage units
  - tablet servers + routers know the partition plan

Why replicas of all data at multiple regions?

- multiple regions -> each user's data geographically close to user
- multiple complete replicas -> maybe survive entire region failure
- complete replicas -> read anything quickly
  - since some data used by many users / many regions
  - once you have multiple regions, fast reads are very important

What are the drawbacks of a copy at each region?

- updates will be slow, need to contact every region
- local reads will probably be stale
- updates from multiple regions need to be sorted out
  - keep replicas identical
  - avoid order anomalies
  - don't lose updates (e.g. read-modify-write for counter)
- disk space probably not an issue for their uses

What is the data and query model?

- basically key/value
- reads/writes probably by column
  - so a write might replace just one column, not whole record
- range scan for ordered tables

How do updates work?

- app server gets web request, needs to write data in PNUTS
- need to update every region!
- why not just have app logic send update to every region?
  - what if app crashes after updating only some regions?
  - what if concurrent updates to same record?

PNUTS has a "record master" for each record

- all updates must go through that region
  - each record has a hidden column indicating region of record master
- responsible storage unit executes updates one at a time per record
- tells MB to broadcast update to all regions
- per-record master probably better than Facebook/memcache master region

So the complete update story (some guesswork):

App wants to update some columns of a record, knows key

1. app sends key and update to local SU1
2. SU1 looks up record master for key: SI2
3. SU1 sends update request to router at SI2
4. router at SI2 forwards update to local SU2 for key
5. SU2 sends update to local Message Broker (MB)
6. MB stores on disk + backup MB, sends vers # to original app how does MB know the vers #? maybe SU2 told it or perhaps SU2 (not MB) replies to original app
7. MB sends update to router at every region
8. every region updates local copy

Puzzles:

- 3.2.1 says MB is commit point
  - i.e. MB writes to log on two disks, keeps trying to deliver why isn't MB disk a terrible bottleneck?
- does update go to MB then SU2? or SU2 then MB? or SU2, MB, SU2?
  - maybe MB then SU2, since MB is commit point
  - maybe SU2 then MB, since SU2 has to check it's the record's master and perhaps pick the new version number, tho maybe not needed
- who replies to client w/ new version #?

All writes are multi-region and thus slow -- why does it make sense?

- application waits for MB commit but not propagation ("asynchronous")
- master likely to be local (they claim 80% of the time)
  - so MB commit will often be quick
  - and app/user will often see its own writes soon
- still, eval says 300ms if master is remote!
- down side: readers at non-master regions may see stale data

How does a read-only query execute?

- multiple kinds of reads (section 2.2) -- why?
- application gets to choose how consistent

- `read-any(k)`
  - read from local SU
  - might return stale data (even if you just wrote!)
  - why: app wants speed but doesn't care about freshness
- `read-critical(k, required_version)`
  - maybe read from local SU if it has vers >= required_version
  - otherwise read from master SU?
  - why: app wants to see its own write
- `read-latest(k)`
  - always read from master SU (? "if local copy too stale")
  - slow if master is remote!
  - why: app needs fresh data

What if app needs to increment a counter stored in a record?

- app reads old value, increments locally, writes new value
- what if the local read produced stale data?
- what if read was OK, but concurrent updates?

`test-and-set-write(version#, new value)` gives you atomic update to one record

- master rejects the write if current version # != version#
- so if concurrent updates, one will lose and retry

`TestAndSet` example:

```
while(1):
  (x, ver) = read-latest(k)
  if(t-a-s-w(k, ver, x+1))
    break
```

The Question

- how does PNUTS cope with Example 1 (page 2)
- Initially Alice's mother is in Alice's ACL, so mother can see photos
  1. Alice removes her mother from ACL
  2. Alice posts spring-break photos
- could her mother see update #2 but not update #1?
  - esp if mother uses different region than Alice or if Alice does the updates from different regions
- ACL and photo list must be in the same record
  - since PNUTS guarantees order only for updates to same record
- Alice sends updates to her record's master region in order

    - master region broadcasts via MB in order
    - MB tells other regions to apply updates in order
- What if Alice's mother:
    - reads the old ACL, that includes mother
    - reads the new photo list
    - answer: just one read of Alice's record, has both ACL and photo list
        - if record doesn't have new ACL, order says it can't have new photos either
- How could a storage system get this wrong?
    - No ordering through single master (e.g. Dynamo)

How to change record's master if no failures?

- e.g. I move from Boston to LA
- perhaps just update the record, via old master?
    - since ID of master region is stored in the record
- old master announces change over MB
- a few subsequent updates might go to the old master
    - it will reject them, app retries and finds new master?

What if we wanted to do bank transfers?

- from one account (record) to another
- can `t-a-s-w` be used for this?
- multi-record updates are not atomic
    - other readers can see intermediate state
    - other writers are not locked out
- multi-record reads are not atomic
    - might read one account before xfer, other account after xfer

Is lack of general transactions a problem for web applications?

- maybe not, if programmers know to expect it

What about tolerating failures?

App server crashes midway through a set of updates

- not a transaction, so only some of writes will happen
- but master SU/MB either did or didn't get each write
    - so each write happens at all regions, or none

SU down briefly, or network temporarily broken/lossy

- (I'm guessing here, could be wrong)
- MB keeps trying until SU acks

    - SU shouldn't ACK until safely on disk

SU loses disk contents, or doesn't automatically reboot

- can apps read from remote regions?
    - paper doesn't say
- need to restore disk content from SUs at other regions
    1. subscribe to MB feed, and save them for now
    2. copy content from SU at another region
    3. replay saved MB updates
- Puzzle:
    - how to ensure we didn't miss any MB updates for this SU?
        - e.g. subscribe to MB at time=100, but source SU only saw through 90?
    - will replay apply updates twice? is that harmful?
    - paper mentions sending checkpoint message through MB
        - maybe fetch copy as of when the checkpoint arrived
        - and only replay after the checkpoint
        - BUT no ordering among MB streams from multiple regions

MB crashes after accepting update

- logs to disks on two MB servers before ACKing
- recovery looks at log, (re)sends logged msgs
- record master SU maybe re-sends an update if MB crash before ACK
    - maybe record version #s will allow SUs to ignore duplicate

MB is a neat idea

- atomic: updates all replicas, or none
    - rather than app server updating replicas (crash...)
- reliable: keeps trying, to cope with temporarily SU/region failure
- async: apps don't have to wait for write to complete, good for WAN
- ordered: keeps replicas identical even w/ multiple writers

Record's master region loses network connection

- can other regions designate a replacement RM?
    - no: original RM's MB may have logged updates, only some sent out
- do other regions have to wait indefinitely? yes
    - this is one price of ordered updates / strict-ish consistency

Evaluation

Evaluation focuses on latency and scaling, not throughput

5.2: time for an insert while busy

- depends on how far away Record Master is
- RM local: 75.6 ms
- RM nearby: 131.5 ms
- RM other coast: 315.5 ms

What is 5.2 measuring? from what to what?

- maybe web server starts insert, to RM replies w/ new version?
- not time for MB to propagate to all regions
    - since then local RM wouldn't be `< remote`

Why 75 ms?

Is it 75 ms of network speed-of-light delay?

- no: local

Is the 75 ms mostly queuing, waiting for other client's operations?

- no: they imply 100 clients was max that didn't cause delay to rise

End of 5.2 suggests 40 ms of 75 ms in in SU

- how could it take 40 ms?
    - each key/value is one file?
    - creating a file takes 3 disk writes (directory, inode, content)?
- what's the other 35 ms?
    - MB disk write?

But only 33 ms (not 75) for "ordered table" (MySQL/Innodb)

- closer to the one or two disk write we'd expect

5.3 / Figure 3: effect of increasing request rate

- what do we expect for graph w/ x-axis req rate, y-axis latency?
    - system has some inherent capacity, e.g. total disk seeks/second
    - for lower rates, constant latency
    - for higher rates, queue grows rapidly, avg latency blows up
- blow-up should be near max capacity of h/w
    - e.g. # disk arms / seek time
- we don't see that in Figure 3
    - apparently their clients were not able to generate too much load
    - end of 5.3 says clients too slow
    - at >= 75 ms/op, 300 clients -> about 4000/sec

- text says max possible rate was about 3000/second
  - 10% writes, so 300 writes/second
  - 5 SU per region, so 60 writes/SU/second
  - about right if each write does a random disk I/O
  - but you'll need lots of SUs for millions of active users

Stepping back, what were PNUTS key design decisions?

1. replication of all data at multiple regions
   - fast reads, slow writes
2. relaxed consistency -- stale reads
   - b/c writes are slow
3. only single-row transactions w/ test-and-set-write
4. sequence all writes thru master region
   - pro: keeps replicas identical, enforces serial order on updates, easy to reason about
   - con: slow, no progress if master region disconnected

Next: Dynamo, a very different design

- async replication, but no master
- eventual consistency
- always allow updates
- tree of versions if network partitions
- readers must reconcile versions

6.824 2015 Lecture 18: Amazon's Dynamo keystore

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Dynamo

- eventually consistent
- considerably less consistent than PNUTS or Spanner
- successful open source projects like Cassandra that have built upon the ideas of Dynamo

Design

- really worried about their service level agreements (SLA)
  - internal SLAs, say between webserver and storage system
- worried about *worst-case* perf., not average perf.
- they want 99.9th percentile of delay `< 300ms`
  - not very clear how this requirement worked itself in the design
  - what choices were made to satisfy this?
- supposed to deal w/ constant failures
  - entire data center offline
- they need the system to be always writeable
  - `=>` no single master

Diagram:

```
Datacenter

    Frontend            Dynamo server
    server
                        Dynamo server
    Frontend
    server              Dynamo server

                        Dynamo server
```

- Guess: amazon has quite a lot of data centers and no one of them are primary or backup, then even if a datacenter goes down only a small fraction of your system is down

    - much more natural to, instead of replicating every record everywhere, to just replicate it on 2 or 3 data centers
- design of Dynamo is not really data-center oriented
- difference from PNUTS is that there's nothing about locality in their design
    - they don't worry about it: no copy of data is made to be near every client
- they need the wide area network to work really well

Details

- always writeable `=>` no single master `=>` different puts on different servers `=>` conflicting updates
- where should puts go and where should gets go so that they are likely to see data written by puts

Consistent hashing

- you hash the key and it tells you what server to put/get it from
- hash output space is a ring/circle
- every key's hash is a point on this circle
- every node's hash is also a point
- `=>` a key will be between nodes or on a node in the circle
- node closest to key on the circle (clockwise) is the key's *coordinator*
- if key is replicated `N` times, then the `N` successor nodes (clockwise) after the key store the key
- even with random choice of node IDs, consistent hashing doesn't uniformly spread the keys across nodes
    - the # of keys on a node is proportional to the gap between that node and its predecessor
    - the distribution of gaps is pretty wide
- to make up for this, virtual nodes are used
    - each physical node is made up of a certain # of virtual nodes, proportional to the perf/capacity of the physical node

Preference lists:

- suppose you have nodes A, B, C, D, E, F and key `k` that hashes before node A
- this key `k` should have 3 copies stored at A, B and C, if `N = 3`
- request could go to the first node A, which could be down
    - or it could go to the first node A, which would try to replicate it on node B and C, which could be down
    - `=>` this first node would replicate on nodes D and E

- `=>` more than `N` nodes that could have the data
- `=>` remember all these nodes in a *preference list*

- request for `k` goes to the first node in the preference list
- that node acts as a coordinator for the request and reads/writes the key on all other nodes
- sloppy quorums,
  - `N` the # of nodes the coordinator sends the request to
  - `R` the # of nodes the coordinator waits for data to come back on a get
  - `W` the # of nodes the coordinator waits for data to write on on a put
- if there are no failures, the coordinator kind of acts like a master
- if there are failures the sloppy quorum makes sure data is persisted, but inconsistencies can be created
- Trouble: because there aren't any real quorums, gets can miss the most recent puts
- you can have nodes A, B, C store some put on stale data, and nodes D, E, F store another put on data
  - the coordinator among D, E, F knows the data is out-of-place and stores a flag to indicate it should be passed to A, B, C (*hinted hand-off*)

Conflicts

- figure 3 in the paper
- when there are 2 conflicting versions, client code has to be able to reconcile them
- dynamo uses version vectors just like Ficus
  - `[a: 1] -> [a: 1, b: 3]`
  - `[a: 1] -> [a: 1, c: 3]`
  - `[a:1, b:3, c: 0]` and `[a:1, b:0, c:3]` conflicts
- Dynamo is weaker than Bayou
- both have a story for how to reconcile conflicted version
  - In dynamo we just have the two conflicting pieces of data, but we don't have the ops that were applied to the state (like remove/add smth from shopping cart)
  - Bayou has the log of the ops
- PNUTS had atomic operation support like a `test-and-set-write` op
  - nothing like that in Dynamo
  - the only way to do that in Dynamo is to be able to merge two conflicting versions

Performance

- *Question to always ask about version vectors:* What happens when the version vectors get too large?

  - they delete entries for nodes that have been modified a long time ago
    - `v1 = [a:1, b:7] -> v1' = [b:7]`
    - what can go wrong? if `[b:7]` is updated to `v2 = [b:8]` then `v2` will conflict with `v1`, even though it was derived directly from it, so the application will get some *false* merges
- they like that they can adjust `N, R, W` to get different trade-offs
  - standard `3,2,2`
  - `3, 3, 1 ->` write quickly but not very durably, reads are rare
  - `3, 1, 3 ->` writes are slow, but reads are quite fast
- the average delays are 5-10ms, much smaller than PNUTS or memcached
  - too small relative to speed-of-light across datacenters
  - but not clear where the data centers were, and what the workloads were

6.824 2015 original notes

```
Dynamo: Amazon's Highly Available Key-value Store
DeCandia et al, SOSP 2007

Why are we reading this paper?
  Database, eventually consistent, write any replica
    Like Ficus -- but a database! A surprising design.
  A real system: used for e.g. shopping cart at Amazon
  More available than PNUTS, Spanner, &c
  Less consistent than PNUTS, Spanner, &c
  Influential design; inspired e.g. Cassandra
  2007: before PNUTS, before Spanner

Their Obsessions
  SLA, e.g. 99.9th percentile of delay < 300 ms
  constant failures
  "data centers being destroyed by tornadoes"
  "always writeable"

Big picture
  [lots of data centers, Dynamo nodes]
  each item replicated at a few random nodes, by key hash

Why replicas at just a few sites? Why not replica at every site?
  with two data centers, site failure takes down 1/2 of nodes
    so need to be careful that *everything* replicated at *both* sites
  with 10 data centers, site failure affects small fraction of nodes
    so just need copies at a few sites

Consequences of mostly remote access (since no guaranteed local copy)
  most puts/gets may involve WAN traffic -- high delays
    maybe distinct Dynamo instances with limited geographical scope?
    paper quotes low average delays in graphs but does not explain
  more vulnerable to network failure than PNUTS
    again since no local copy

Consequences of "always writeable"
  always writeable => no master! must be able to write locally.
  always writeable + failures = conflicting versions

Idea #1: eventual consistency
  accept writes at any replica
  allow divergent replicas
  allow reads to see stale or conflicting data
```

```
    resolve multiple versions when failures go away
      latest version if no conflicting updates
      if conflicts, reader must merge and then write
    like Bayou and Ficus -- but in a DB

  Unhappy consequences of eventual consistency
    May be no unique "latest version"
    Read can yield multiple conflicting versions
    Application must merge and resolve conflicts
    No atomic operations (e.g. no PNUTS test-and-set-write)

  Idea #2: sloppy quorum
    try to get consistency benefits of single master if no failures
      but allows progress even if coordinator fails, which PNUTS does not
    when no failures, send reads/writes through single node
      the coordinator
      causes reads to see writes in the usual case
    but don't insist! allow reads/writes to any replica if failures

  Where to place data -- consistent hashing
    [ring, and physical view of servers]
    node ID = random
    key ID = hash(key)
    coordinator: successor of key
      clients send puts/gets to coordinator
    replicas at successors -- "preference list"
    coordinator forwards puts (and gets...) to nodes on preference list

  Why consistent hashing?
    Pro
      naturally somewhat balanced
      decentralized -- both lookup and join/leave
    Con (section 6.2)
      not really balanced (why not?), need virtual nodes
      hard to control placement (balancing popular keys, spread over sites)
      join/leave changes partition, requires data to shift

  Failures
    Tension: temporary or permanent failure?
      node unreachable -- what to do?
      if temporary, store new puts elsewhere until node is available
      if permanent, need to make new replica of all content
    Dynamo itself treats all failures as temporary

  Temporary failure handling: quorum
    goal: do not block waiting for unreachable nodes
    goal: put should always succeed
    goal: get should have high prob of seeing most recent put(s)
    quorum: R + W > N
      never wait for all N
      but R and W will overlap
      cuts tail off delay distribution and tolerates some failures
    N is first N *reachable* nodes in preference list
      each node pings successors to keep rough estimate of up/down
      "sloppy" quorum, since nodes may disagree on reachable
    sloppy quorum means R/W overlap *not guaranteed*

  coordinator handling of put/get:
    sends put/get to first N reachable nodes, in parallel
    put: waits for W replies
    get: waits for R replies
    if failures aren't too crazy, get will see all recent put versions

  When might this quorum scheme *not* provide R/W intersection?

  What if a put() leaves data far down the ring?
    after failures repaired, new data is beyond N?
    that server remembers a "hint" about where data really belongs
    forwards once real home is reachable
    also -- periodic "merkle tree" sync of key range

  How can multiple versions arise?
```

```
  Maybe a node missed the latest write due to network problem
  So it has old data, should be superseded by newer put()s
  get() consults R, will likely see newer version as well as old

How can *conflicting* versions arise?
  N=3 R=2 W=2
  shopping cart, starts out empty ""
  preference list n1, n2, n3, n4
  client 1 wants to add item X
    get(cart) from n1, n2, yields ""
    n1 and n2 fail
    put(cart, "X") goes to n3, n4
  client 2 wants to delete X
    get(cart) from n3, n4, yields "X"
    put(cart, "") to n3, n4
  n1, n2 revive
  client 3 wants to add Y
    get(cart) from n1, n2 yields ""
    put(cart, "Y") to n1, n2
  client 3 wants to display cart
    get(cart) from n1, n3 yields two values!
      "X" and "Y"
      neither supersedes the other -- the put()s conflicted

How should clients resolve conflicts on read?
  Depends on the application
  Shopping basket: merge by taking union?
    Would un-delete item X
    Weaker than Bayou (which gets deletion right), but simpler
  Some apps probably can use latest wall-clock time
    E.g. if I'm updating my password
    Simpler for apps than merging
  Write the merged result back to Dynamo

How to detect whether two versions conflict?
  As opposed to a newer version superseding an older one
  If they are not bit-wise identical, must client always merge+write?
  We have seen this problem before...

Version vectors
  Example tree of versions:
    [a:1]
           [a:1,b:2]
    VVs indicate v1 supersedes v2
    Dynamo nodes automatically drop [a:1] in favor of [a:1,b:2]
  Example:
    [a:1]
           [a:1,b:2]
    [a:2]
    Client must merge

get(k) may return multiple versions, along with "context"
  and put(k, v, context)
  put context tells coordinator which versions this put supersedes/merges

Won't the VVs get big?
  Yes, but slowly, since key mostly served from same N nodes
  Dynamo deletes least-recently-updated entry if VV has > 10 elements

Impact of deleting a VV entry?
  won't realize one version subsumes another, will merge when not needed:
    put@b: [b:4]
    put@a: [a:3, b:4]
    forget b:4: [a:3]
    now, if you sync w/ [b:4], looks like a merge is required
  forgetting the oldest is clever
    since that's the element most likely to be present in other branches
    so if it's missing, forces a merge
    forgetting *newest* would erase evidence of recent difference

Is client merge of conflicting versions always possible?
  Suppose we're keeping a counter, x
```

```
    x starts out 0
    incremented twice
    but failures prevent clients from seeing each others' writes
    After heal, client sees two versions, both x=1
    What's the correct merge result?
    Can the client figure it out?

  What if two clients concurrently write w/o failure?
    e.g. two clients add diff items to same cart at same time
    Each does get-modify-put
    They both see the same initial version
    And they both send put() to same coordinator
    Will coordinator create two versions with conflicting VVs?
      We want that outcome, otherwise one was thrown away
      Paper doesn't say, but coordinator could detect problem via put() context

  Permanent server failures / additions?
    Admin manually modifies the list of servers
    System shuffles data around -- this takes a long time!

  The Question:
    It takes a while for notice of added/deleted server to become known
      to all other servers. Does this cause trouble?
    Deleted server might get put()s meant for its replacement.
    Deleted server might receive get()s after missing some put()s.
    Added server might miss some put()s b/c not known to coordinator.
    Added server might serve get()s before fully initialized.
    Dynamo probably will do the right thing:
      Quorum likely causes get() to see fresh data as well as stale.
      Replica sync (4.7) will fix missed get()s.

  Is the design inherently low delay?
    No: client may be forced to contact distant coordinator
    No: some of the R/W nodes may be distant, coordinator must wait

  What parts of design are likely to help limit 99.9th pctile delay?
    This is a question about variance, not mean
    Bad news: waiting for multiple servers takes *max* of delays, not e.g. avg
    Good news: Dynamo only waits for W or R out of N
      cuts off tail of delay distribution
      e.g. if nodes have 1% chance of being busy with something else
      or if a few nodes are broken, network overloaded, &c

  No real Eval section, only Experience

  How does Amazon use Dynamo?
    shopping cart (merge)
    session info (maybe Recently Visited &c?) (most recent TS)
    product list (mostly r/o, replication for high read throughput)

  They claim main advantage of Dynamo is flexible N, R, W
    What do you get by varying them?
    N-R-W
    3-2-2 : default, reasonable fast R/W, reasonable durability
    3-3-1 : fast W, slow R, not very durable, not useful?
    3-1-3 : fast R, slow W, durable
    3-3-3 : ??? reduce chance of R missing W?
    3-1-1 : not useful?

  They had to fiddle with the partitioning / placement / load balance (6.2)
    Old scheme:
      Random choice of node ID meant new node had to split old nodes' ranges
      Which required expensive scans of on-disk DBs
    New scheme:
      Pre-determined set of Q evenly divided ranges
      Each node is coordinator for a few of them
      New node takes over a few entire ranges
      Store each range in a file, can xfer whole file

  How useful is ability to have multiple versions? (6.3)
    I.e. how useful is eventual consistency
    This is a Big Question for them
```

```
  6.3 claims 0.001% of reads see divergent versions
    I believe they mean conflicting versions (not benign multiple versions)
    Is that a lot, or a little?
  So perhaps 0.001% of writes benefitted from always-writeable?
    I.e. would have blocked in primary/backup scheme?
  Very hard to guess:
    They hint that the problem was concurrent writers, for which
      better solution is single master
    But also maybe their measurement doesn't count situations where
      availability would have been worse if single master

Performance / throughput (Figure 4, 6.1)
  Figure 4 says average 10ms read, 20 ms writes
    the 20 ms must include a disk write
    10 ms probably includes waiting for R/W of N
  Figure 4 says 99.9th pctil is about 100 or 200 ms
    Why?
    "request load, object sizes, locality patterns"
    does this mean sometimes they had to wait for coast-coast msg?

Puzzle: why are the average delays in Figure 4 and Table 2 so low?
  Implies they rarely wait for WAN delays
  But Section 6 says "multiple datacenters"
    you'd expect *most* coordinators and most nodes to be remote!
    Maybe all datacenters are near Seattle?

Wrap-up
  Big ideas:
    eventual consistency
    always writeable despite failures
    allow conflicting writes, client merges
  Awkward model for some applications (stale reads, merges)
    this is hard for us to tell from paper
  Maybe a good way to get high availability + no blocking on WAN
    but PNUTS master scheme implies Yahoo thinks not a problem
  No agreement on whether eventual consistency is good for storage systems
```

6.824 2015 Lecture 19: HubSpot

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Distributed systems in the real world

Who builds distributed systems:

- SaaS market
  - Startups: CustomMade, Instagram, HubSpot
  - Mature: Akamai, Facebook, Twitter
- Enterprise market
  - Startup: Basho (Riak), Infinio, Hadapt
  - Mature: VMWare, Vertica
- ...and graduate students

High-level components:

- front-end: load balancing routers
- handlers, caching, storage, business services
- infra-services: logging, updates, authentication

Low-level components:

- RPCs (semantics, failure)
- coordination (consensus, Paxos)
- persistence (serialization semantics)
- caching
- abstractions (queues, jobs, workflows)

Building the thing

Business needs will affect scale and architecture

- dating website core data: OkCupid uses 2 beefy database servers
- analytics distributed DB: Vertica/Netezza clusters have around 100 nodes
- mid-size SaaS company: HubSpot uses around 100 single-node DBs or around 10 node HBase clusters
  - MySQL mostly

- Akamai, Facebook, Amazon: tens of thousands of machines

Small SaaS startup:

- early on the best thing is to figure out if you have a good idea that people would buy
- typically use a platform like Heroku, Google App Engine, AWS, Joyent, CloudFoundry

Midsized SaaS:

- need more control than what PaaS offers
- scale may enable you to build better solutions more cheaply
- open source solutions can help you

Mature SaaS:

- Jepsen tool
- "Ensure your design works if scale changes by 10x or 20x; the right solution for x often not optimal for 100x", Jeff Dean

How to think about your design:

- understand what your system needs to do and the semantics
- understand workload scale then estimate (L2 access time, network latency) and plan to understand performance

Running the thing

- "telemetry beats event logging"
  - logs can be hard to understand: getting a good story out is difficult
- logging: first line of defense, doesn't scale well
  - logs on different machines
  - what if timestamps are useless because clocks are not synced
  - lots of tools around logging
  - having log data in queryable format tends to be very useful
- monitoring, telemetry, alerting
  - annotate code with timing and counting events
  - measure how big a memory queue is or how long a request takes and you can count it
  - can do telemetry at multiple granularities so we can break long requests into smaller pieces and pinpoint problems

Management: command and control

- in classroom settings you don't have to set up a bunch of machines
- as your business scales new machines need to be set up => must automate
- separate configuration from app
- HubSpot uses a ZooKeeper like system that allows apps to get config values
- Maven for dependencies in Java
- Jenkins for continuous integration testing

Testing

- automated testing makes it easy to verify newly introduced changes to your code
- UI testing can be a little harder (simulate clicks, different layout in different browsers)
    - front end changes => must change tests?

Teams

- people: how do you get together and build the thing
- analogy: software engineering process is sort of like a distributed system with unreliable components.
    - somehow must build reliable software on a reliable schedule
- gotta take care of your people: culture has to be amenable to people growing, learning and failing

Process

- waterfall: big design upfront and then implement it
- agile/scrum: don't know the whole solution, need to iterate on designs
- kanban:
- lean:

Questions

- making a big change on fast changing code base
    - if you branch and then merge your changes, chances are the codebase has changed drastically
    - you can try to have two different branches deployed such that the new branch can be tested in production
- culture changes with growth
    - need to pay attention to culture and happiness of employees

- very important to measure happiness
- having small teams might help because people can own projects

6.824 2015 Lecture 20: Argus

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Atomic commit: two-phase commit

- how to use two-phase commit for distributed transactions
- Argus

You have a bunch of computers that do different things (not replicas). Like two computers, one stores events for people in A-L, another for people in M-Z. If you want to create an event for Alice and Mike you need to interact with both servers and make sure that the event is either created on both or on neither.

The challenges are *crashes* and *network failures* which inject ambiguities (not responding cause of crash or network failure?)

In Ivy and TreadMarks if one of the machines crashed it had no way to recover. We also saw MapReduce and Spark which had a story for crash recovery.

Code:

```
schedule(u1 user, u2 user, t time):
    ok1 = reserve(u1, t)    # reserve for the 1st user
    ok2 = reserve(u2, t)    # reserve for the 2nd user

    # Tricky: if the 1st reserve succeeded and the 2nd didn't => trouble
    # We'd like to deal with this in the following way:

    if ok1 and ok2
        commit
    else
        abort

    # One bad way to make this work is to let the servers chit-chat and
    # make sure they both committed.
    # At no stage in a transaction like this can the servers finish
    #   - S1: I'll do it if you do it
    #   - S2: I'll do it if you do it
    #   - S1: I'll do it if you do it
    #   - S2: I'll do it if you do it
    # (sounds like the two generals problem?)
```

Idea 1: tentative changes

```
reserve(u user, t time):
    if u[t] = free          # if user's calendar is free at time t
        tent[t] = taken     # ...then tenatively schedule
commit:
    copy tent[t] to u[t]
abort:
    discard tent[t]
```

Idea 2: single machine/entity (transaction coordinator) decides

```
client           TC      A       B
    \-------------------->
    \---------------------------->
                         |       |
    <--------------------/       |
    <----------------------------/
                ------------
    ----- GO ----> |          |
                 | |          |
    <------------/ |          |
                ------------
```

Properties:

- state: unknown, committed, aborted
- if any thinks "committed", then none think "aborted"
- if any think "aborted", then none think "committed"

Two-phase commit (2PC)

Used frequently in real distributed databases.

```
client          TC          A           B
                    .
                    .
                    .
    ---- GO ---->                             --\
                   prepare                      |
                ------------>                   |
                ------------------------>       | Phase 1
                   yes/no   |           |       |
                <-----------/           |       |
                <-----------------------/     --/

                   commit/abort
                ------------>                 --\
                ------------------------>       | Phase 2
    commit/abort                                |
  <-------------                              --/
```

`Prepare` asks "are you still alive and willing to commit this transaction?"

- servers may say no

- servers may be unreachable

Termination protocol

- maybe the TC has a timeout while it's waiting for the yes/no response to one or more prepare messages
  - at this point, it can abort the transaction, because no one has started a commit (since the TC did not send it, since it was waiting on yes/no)
- B times out while waiting for prepare message
  - `=>` B hasn't replied to prepare `=>` TC hasn't sent commit to the participants `=>` TC can send abort
- B times out waiting for commit/abort after saying *no* to prepare
  - `=>` B can abort because it knows the TC will abort everyone
- B times out waiting for commit/abort after saying *yes* to prepare
  - `=>` B said yes to TC and TC could have received `yes` from everyone else (or not) `=>` outcome can be either commit or abort `=>` B has to wait
  - there are some lucky cases in which `B` could decide to abort/commit if `A` tells it via another channel

Does this waiting make 2PC impractical? People are split up?

What about reboots? If one of the participants said yes to a prepare, it has to remember that across reboots or crashes, so that it can be able to finish the transaction (commit or abort).

- in the calendar example, it would also need to remember the tentative schedule in `tent[]`
- extra note: since in the diagram the TC did not wait for ACKs on commit/abort the participants need to persist their locks around the transaction so that they don't do a subsequent transaction before this one is finished

What happens if TC crashes in the middle of sending commits?

- it has to remember all committed/non-committed transactions

Resemblance to Paxos?

- Paxos is a way to build highly available systems by replication (all servers have all data and are doing the same thing)
  - Paxos system can proceed even if some of the servers are down
- 2PC you cannot make progress even if just one server is down
  - each server is doing a *different thing* (want every server to do its own part in a transaction)
- While 2PC helps a set of servers reach agreement, it it not fault tolerant or available (it cannot proceed when servers are down)

- You might think you can do the calendar scheduling with Paxos by having both servers agree on the schedule op. However, while agreeing on the op will work, committing the op will not. For instance, what if one server's user is busy during the scheduled time? Then it cannot commit the op while the other one might be able to. Paxos doesn't help solve that conflict.

Atomic distributed transactions: write your transaction code without thinking about what other transactions could be going on

Bank example:

```
T1:
    addToBal(x, 1)
    addToBal(y, -1)

    # Need this to be a transaction to implement a transfer correctly

T2:
    tmp1 = getBal(x)
    tmp2 = getBal(y)

    print(tmp1, tmp2)

    # We cannot have the execution of T1 interleave with the execution of
    # T2. T2 had better see both addToBal calls or no addToBal calls from T1
```

This is called *serializability*: The effect of running a bunch of transactions is the same as if they were run in some sequential order (no interleaving allowed: exec first half of T1, exec first half of T2, finish second half of T1, finish T2).

One way to implement transactions is to use locks for each data record that are acquired before a transaction begins operating on those records and holds them until it commits or aborts. This is called **two-phase locking**.

Deadlock can occur if T1 acquires x and then y while T2 acquires y and then x. Database systems for instance have ways to deal with this::

- timeout on acquiring locks and retry
- only allow transactions to acquire locks in a certain order
- perform deadlock detection if single-machine setup

Nobody ever likes to use 2PC.

- because of the waiting/blocking issue when a server times out waiting for a commit/abort after having said "no" to a prepare

When participants acquire locks they are holding them across multiple RTTs in the network because you have to wait for the commit message.

Argus

- the cool thing is that it attempts to absorb as much of the nitty-gritty junk of distributed systems programming inside the language
- the desire was to have a clean story for handling RPC failures
- Argus sets up a framework where RPC failures can be handled cleanly
    - does all the bookkeeping required to rollback the transactions
- Argus has to know about the data in order to be able to rollback
    - it needs to create tentative updates and so on

6.824 notes

```
6.824 2015 Lecture 20: Two-Phase Commmit

Topics:
  distributed commit, two-phase commit
  distributed transactions
  Argus -- language for distributed programming

Distributed commit:
  A bunch of computers are cooperating on some task, e.g. bank transfer
  Each computer has a different role, e.g. src and dst bank account
  Want to ensure atomicity: all execute, or none execute
    "distributed transaction"
  Challenges: crashes and network failures

Example:
  calendar system, each user has a calendar
  want to schedule meetings with multiple participants
  one server holds calendars of users A-M, another server holds N-Z
  [diagram: client, two servers]
  sched(u1, u2, t):
    begin_transaction
      ok1 = reserve(u1, t)
      ok2 = reserve(u2, t)
      if ok1 and ok2:
        commit
      else
        abort
    end_transaction
  the reserve() calls are RPCs to the two calendar servers
  We want both to reserve, or both not to reserve.
  What if 1st reserve() returns true, and then:
    2nd reserve() returns false (time not available)
    2nd reserve() doesn't return (lost RPC msg, u2's server crashes)
    2nd reserve() returns but then crashes
    client fails before 2nd reserve()
  We need a "distributed commit protocol"

Idea: tentative changes, later commit or undo (abort)
  reserve_handler(u, t):
    if u[t] is free:
      temp_u[t] = taken -- A TEMPORARY VERSION
      return true
    else:
      return false
  commit_handler():
    copy temp_u[t] to real u[t]
  abort_handler():
    discard temp_u[t]
```

```
Idea: single entity decides whether to commit
  to prevent any chance of disagreement
  let's call it the Transaction Coordinator (TC)
  [time diagram: client, TC, A, B]
  client sends RPCs to A, B
  on end_transaction, client sends "go" to TC
  TC/A/B execute distributed commit protocol...
  TC reports "commit" or "abort" to client

We want two properties for distributed commit protocol:
  TC, A, and B start in state "unknown"
    each can move to state "abort" or "commit"
    but then each never changes mind
  Correctness:
    if any commit, none abort
    if any abort, none commit
  Performance:
    (since doing nothing is correct...)
    if no failures, and A and B can commit, then commit.
    if failures, come to some conclusion ASAP.

We're going to develop a protocol called "two-phase commit"
  Used by distributed databases for multi-server transactions
  And by Spanner and Argus

Two-phase commit without failures:
  [time diagram: client, TC, A, B]
  client sends reserve() RPCs to A, B
  client sends "go" to TC
  TC sends "prepare" messages to A and B.
  A and B respond, saying whether they're willing to commit.
    Respond "yes" if haven't crashed, timed out, &c.
  If both say "yes", TC sends "commit" messages.
  If either says "no", TC sends "abort" messages.
  A/B "decide to commit" if they get a commit message.
    I.e. they actually modify the user's calendar.

Why is this correct so far?
  Neither can commit unless they both agreed.
  Crucial that neither changes mind after responding to prepare
    Not even if failure

What about failures?
  Network broken/lossy
  Server crashes
  Both visible as timeout when expecting a message.

Where do hosts wait for messages?
  1) TC waits for yes/no.
  2) A and B wait for prepare and commit/abort.

Termination protocol summary:
  TC t/o for yes/no -> abort
  B t/o for prepare, -> abort
  B t/o for commit/abort, B voted no -> abort
  B t/o for commit/abort, B voted yes -> block

TC timeout while waiting for yes/no from A/B.
  TC has not sent any "commit" messages.
  So TC can safely abort, and send "abort" messages.

A/B timeout while waiting for prepare from TC
  have not yet responded to prepare
  so can abort
  respond "no" to future prepare

A/B timeout while waiting for commit/abort from TC.
  Let's talk about just B (A is symmetric).
  If B voted "no", it can unilaterally abort.
  So what if B voted "yes"?
  Can B unilaterally decide to abort?
```

```
    No! TC might have gotten "yes" from both,
    and sent out "commit" to A, but crashed before sending to B.
    So then A would commit and B would abort: incorrect.
  B can't unilaterally commit, either:
    A might have voted "no".

If B voted "yes", it must "block": wait for TC decision.

What if B crashes and restarts?
  If B sent "yes" before crash, B must remember!
    --- this is today's question
  Can't change to "no" (and thus abort) after restart
  Since TC may have seen previous yes and told A to commit
  Thus:
    B must remember on disk before saying "yes", including modified data.
    B reboots, disk says "yes" but no "commit", must ask TC.
    If TC says "commit", copy modified data to real data.

What if TC crashes and restarts?
  If TC might have sent "commit" or "abort" before crash, TC must remember!
    And repeat that if anyone asks (i.e. if A/B/client didn't get msg).
    Thus TC must write "commit" to disk before sending commit msgs.
  Can't change mind since A/B/client have already acted.

This protocol is called "two-phase commit".
  What properties does it have?
  * All hosts that decide reach the same decision.
  * No commit unless everyone says "yes".
  * TC failure can make servers block until repair.

What about concurrent transactions?
  We realy want atomic distributed transactions,
    not just single atomic commit.
  x and y are bank balances
  x and y start out as $10
  T1 is doing a transfer of $1 from x to y
  T1:
    add(x, 1)  -- server A
    add(y, -1) -- server B
  T2:
    tmp1 = get(x)
    tmp2 = get(y)
    print tmp1, tmp2

Problem:
  what if T2 runs between the two add() RPCs?
  then T2 will print 11, 10
  money will have been created!
  T2 should print 10,10 or 9,11

The traditional approach is to provide "serializability"
  results should be as if transactions ran one at a time in some order
  either T1, then T2; or T2, then T1

Why serializability?
  it allows transaction code to ignore the possibility of concurrency
  just write the transaction to take system from one legal state to another
  internally, the transaction can temporarily violate invariants
    but serializability guarantess no-one will notice

One way to implement serializabilty is with "two-phase locking"
  this is what Argus does
  each database record has a lock
  the lock is stored at the server that stores the record
    no need for a central lock server
  each use of a record automatically acquires the record's lock
    thus add() handler implicitly acquires lock when it uses record x or y
  locks are held until *after* commit or abort

Why hold locks until after commit/abort?
  why not release as soon as done with the record?
  e.g. why not have T2 release x's lock after first get()?
```

```
    T1 could then execute between T2's get()s
    T2 would print 10,9
    but that is not a serializable execution: neither T1;T2 nor T2;T1

2PC perspective
  Used in sharded DBs when a transaction uses data on multiple shards
  But it has a bad reputation:
    slow because of multiple phases / message exchanges
    locks are held over the prepare/commit exchanges
    TC crash can cause indefinite blocking, with locks held
  Thus usually used only in a single small domain
    E.g. not between banks, not between airlines, not over wide area

Paxos and two-phase commit solve different problems!
  Use Paxos to high availability by replicating
    i.e. to be able to operate when some servers are crashed
    the servers must have identical state
  Use 2PC when each participant does something different
    And *all* of them must do their part
  2PC does not help availability
    since all servers must be up to get anything done
  Paxos does not ensure that all servers do something
    since only a majority have to be alive

What if you want high availability *and* distributed commit?
  [diagram]
  Each "server" should be a Paxos-replicated service
  And the TC should be Paxos-replicated
  Run two-phase commit where each participant is a replicated service
  Then you can tolerate failures and still make progress
  This is what Spanner does (for update transactions)

Case study: Argus

Argus's big ideas:
  Language support for distributed programs
    Very cool: language abstracts away ugly parts of distrib systems
    Aimed at different servers doing different jobs, cooperating
  Easy fault tolerance:
    Transactional updates
    So crash results in entire transaction un-done, not partial update
  Easy persistence ("stable"):
    Ordinary variables automatically persisted to disk
    Automatic crash recovery
  Easy concurrency:
    Implicit locking of language objects
  Easy RPC model:
    Method calls transparently turned into RPCs
    RPC failure largely hidden via transactions, two-phase commit

We've seen the fundamental problem before
  What to do if *part* of a distributed computation crashes?
  IVY/Treadmarks had no answer
  MR/Spark could re-execute *part* of computation, for big data

Picture
  "guardian" is like an RPC server
    has state (variables) and handlers
  "handler" is an RPC handler
    reads and writes local variables
  "action" is a distributed atomic transaction
  action on A
    A RPC to B
      B RPC to C
    A RPC to D
  A finishes action
    prepare msgs to B, C, D
    commit msgs to B, C, D

The style is to send RPC to where the data is
  Not to fetch the data
  Argus is not a storage system
```

```
Look at bank example
  page 309 (and 306): bank transfer

Points to notice
  stable keyword (programmer never writes to disk &c)
  atomic keyword (programmer almost never locks/unlocks)
  enter topaction (in transfer)
  coenter (in transfer)
  RPCs are hidden (e.g. f.withdraw())
  RPC error handling hidden (just aborts)

what if deposit account doesn't exist?
  but f.withdraw(from) has already been called?
  how to un-do?
  what's the guardian state when withdraw() handler returns?
    lock, temporary version, just in memory

what if an audit runs during a transfer?
  how does the audit not see the tentative new balances?

if a guardian crashes and reboots, what happens to its locks?
  can it just forget about pre-crash locks?

subactions
  each RPC is actually a sub-action
  the RPC can fail or abort w/o aborting surrounding action
  this lets actions e.g. try one server, then another
  if RPC reply lost, subaction will abort, undo
    much cleaner than e.g. Go RPC

is Argus's implicit locking the right thing?
  very convenient!
  don't have to worry about forgetting to lock!
  (though deadlocks are easy)
  databases work (and worked) this way; it's a sucessful idea

is transactions + RPC + 2PC a good design point?
  programmability pro:
    very easy to get nice fault tolerance semantics
  performance con:
    lots of msgs and disk writes
    2PC and 2PL hold locks for a while, block if failure

is Argus's language integration the right thing?
  i.e. persisting and locking language objects
  it looks very convenient (and it is)
  but it turns out to be even more valuable have relational tables
    people like queries/joins/&c over tables, rows, columns
    that is, people like a storage abstraction!
  maybe there is a better language-based scheme waiting to be found
```

6.824 2015 Lecture 21: Optimistic concurrency control, Thor

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Optimistic concurrency control

- want to build stable storage systems with high-speed, large-scale, and good semantics
  - struggle to get them all
- Thor is another tool that might help us build such a system

Diagram:

```
  Client              Client
 -----------         -----------
| App       |       | App       |
 -----------         -----------     ......
| Thor cache|       | Thor cache|
|           |       |           | ------
 ----------- <-  --> -----------        \
   \ write x   \/                        \   read-set
    |          / invalidate messages      \  write-set
   \|/        /                            \
    .        /                             \/
  Server    /         Server          Validation service
 -----------         -----------         -----------
| App       |       | App       |       |           |
 -----------         -----------         -----------
| Data      |       | Data      |
|   A-M     |       |   N-Z     |
 -----------         -----------
```

If you step back a little, this looks a lot like the Facebook/memcache paper where data is sharded and caches are used on the client side.

- want to be able to support concurrent transactions
- caching makes transactions tricky: could be reading stale data from cache
- locking can solve all of our problems
  - but application would have to chat to server every time it wants an item `=>` defeats the purpose of a cache

Thor uses optimistic concurrency control: the application goes ahead, doing its reads and writes on whatever data happens to be available in the cache. When the transaction wants to commit, it has to talk to some validation service, who will look at the order in which writes and reads happened.

- optimistic in the sense that transactions are executing, ignoring other tx's and hoping it didn't conflict with anything, later checking if it did

Validation scheme 1

- assuming a single valdiation server
- the read set and the write set that clients send for validation are actual values that the transactions read/wrote
- the scheme takes the descriptions of the transactions, try all possible orders and see if any order results in a sequentially-consistent order
- validator doesn't know what the transaction did internally

Example 1:

```
x = y = z = 0

# Then, 4 tx's are run

T1: Rx0 Wx1
T2: Rz0 Wz9
T3: Ry1 Rx1
T4: Rx0 Wy1
```

For instance T3, T1 not possible:

But `T4 (Rx0 Wy1), T1 (Rx0 Wx1), T3 (Ry1 Rx1), T2(Rz0 Wz9)` is possible: the order is consistent with the values read by the transactions.

- Note: if transactions came from different machines (which obviously don't share caches/communicate), then the question arises of how T3 read the value 1 for x, when the DB was initialized to 0? The answer is "this is just a made up example: imagine they did actually share a cache. the point is to see that the validator orders the transactions so that they are consistent"
- Note that T2 only read/wrote `z =>` did not conflict with any of `T1, T3, T4`
  - in situations like these, OCC performs very well

This scheme is great because it allows us to execute transactions without locking.

Example 2:

```
x = y = 0

T1: Rx0 Wx1
T2: Rx0 Wy=1
T3: Ry0 Rx=1
```

This 3 transactions cannot be serialized:

- T1 --- before --> T3 (t3 read x1, t1 wrote x1)
- T3 --- before --> T2 (t3 read y0, t2 wrote y1)
- T2 --- before --> T1 (t2 read x0, t1 wrote x1)
  - cycle! `=>` cannot serialize

Another thing to ask of any OCC scheme is whether it can cleverly handle *read-only transactions* (some schemes can).

- Thor and the schemes talked about today do have to validate read only tx's

Example 3:

```
x = y = z = 0

T1 Wx1
T2 Rx1 Wy2
T3 Ry2 Rx0

T3 read x=0 => T3 comes before T1
T3 read y=2 => T3 comes after T2
T2 read x=1 => T2 comes after T1

  /---------\
 /          \/
T2 <- T3 <- T1
```

If we version records, and make sure that read-only tx's only read the same version of records => we can place it anywhere in the sequence of serialized transactions after that version => can be serialized

Why not use read-write locks?

- simple transactions like `x=x+1` first acquire a read lock and then need to upgrade it to a write lock
  - if you have two such transactions `=>` deadlock because none will release its read lock until it upgrades to the write lock

Distributed validation

If we have data sharded on more than one server, then server 1 (A-M) can just validate part of the transaction that affects records A-M and server 2 can look at the part that affects records N-Z. Then the clients can make sure using two-phase commit (2PC) that the two servers both okayed the transaction.

A naive implementation like this will not work though:

Example 2 (from before):

```
x on server 1
y on server 2

    sv1         sv2
  ---------- ---------
T1 Rx0 Wx1  |
T2 Rx0      |   Wy1
T3 Rx1      |   Ry0

sv1: T2 T1 T3 (yes)
sv2: T3 T2 (yes)
```

But, the result is incorrect because the validators are saying "yes" to different orders.

To solve this, see validation scheme 2

Validation scheme 2

Use timestamps to build a working distributed validation scheme

Every time a client would like to commit a transaction, the server choses a timestamp for this transaction based on its local clock (*loosely* synchronized to the real time).

Validation simply checks that the order implied by the timestamps is consistent with the reads and writes that the transactions performed.

Example 2 (again)

```
                        sv1     |   sv2
T1@100: Rx0 Wx1       Rx0 Wx1   |
T2@110: Rx0 Wy1       Rx0       |   Wy1
T3@105: Ry0 Rx1       Rx1       |   Ry0
                    T1 T3 T2       T2 T3
                      (no)          (no)
```

Loosely synchronized clocks `=>` Have to be prepared to deal with T2 running before T3 even though timestamp says it runs later.

The only thing that the timestamps give the servers is the ability to agree on an order for the transactions amongst themselves so that they don't say yes to different orders.

- `=>` forced to use the timestamp order (cannot search for better order, even though it might be possible)
    - but the distributed algorithm for validation is very straightforward

Example:

```
T1@100: Rx0 Wx1
T2@ 90: Rx1 Wx2  (assigned lower timestamp due to loose sync)
```

This could have committed in the previous scheme (there is a valid order T1, T2) but this current scheme restricts the order to be `T2, T1` due to the timestamps.

- this is just a performance problem of course

Trouble: The read sets and write sets look at values, to check for conflicting transactions. That gives the validator some power (ability to commit more tx's) but is impractical when the values are big.

- `=>` use a version number instead of values

Example:

```
T1: Wx1 v105
T2: Wx2 v106
T3: Wx1 v107
T4: Rx1 v105
-> aborted becauses T4 read stale version of x (105)
 -> however, even though it read the v105 it still got the same value as
    the freshest version v107 (the value 1) => unnecessary abort
```

So now version numbers need to be stored next to every record. Thor doesn't want to do this. Instead Thor shoots down transactions that read stale records by sending invalidate messages when those records are written. Clients who cached those records can then discard them.

6.824 notes

```
Paper: Efficient Optimistic Concurrency Control using Loosely
Synchronized Clocks, by Adya, Gruber, Liskov and Maheshwari.

Why this paper?
  to look at optimistic concurrency control (OCC)
  OCC might help us get large scale, high speed, *and* good semantics

Thor overview
  [clients, client caches, servers A-M N-Z]
  data sharded over servers
  code runs in clients (not like Argus; not an RPC system)
  clients read/write DB records from servers
  clients cache data locally for fast access
    on client cache miss, fetch from server

Thor arrangement is fairly close to modern big web site habits
  clients, local fast cache, slower DB servers
  like Facebook/memcache paper
  but Thor has much better semantics

Thor programs use fully general transactions
  multi-operation
  serializable
  so can do bank xfers w/o losing money, &c

Client caching makes transactions tricky
  writes have to invalidatate cached copies
  how to cope with reads of stale cached data?
```

```
  how to cope with read-modify-write races?
  clients could lock before using each record
    but that's slow -- probably need to contact server
    wrecks the whole point of fast local caching in clients
    (though caching read locks might be OK, as in paper Eval)

Thor uses optimistic concurrency control (OCC)
  an idea from the early 1980s
  just read and write the local copy
    don't worry about other transactions until commit
  when transaction wants to commit:
    send read/write info to server for "validation"
    validation decides if OK to commit -- if serializable
    if yes, send invalidates to clients with cached copies of written records
    if no, abort, discard writes
  optimistic b/c hopes for no conflict
    if turns out to be true, fast!
    if false, validation can detect, but slow

What should validation do?
  it looks at what the executing transactions read and wrote
  decides if there's a serial execution order that would have gotten
    the same results as the actual concurrent execution
  there are many OCC validation algorithms!
    i will outline a few, leading up to Thor's

Validation scheme #1
  a single validation server
  clients tell validation server the read and write VALUES
    seen by each transaction that wants to commit
    "read set" and "write set"
  validation must decide:
    would the results be serializable if we let these
      transactions commit?
  scheme #1 shuffles the transactions, looking for a serial order
    in which each read sees the value written by the most
    recent write; if one exists, the execution was serializable.

Validation example 1:
  initially, x=0 y=0 z=0
  T1: Rx0 Wx1
  T2: Rz0 Wz9
  T3: Ry1 Rx1
  T4: Rx0 Wy1
  validation needs to decide if this execution (reads, writes)
    is equivalent to some serial order
  yes: one such order is T4, T1, T3, T2; says yes to all
    (really T2 can go anywhere)
  note this scheme is far more permissive than Thor's
    e.g. it allows transactions to see uncommitted writes

OCC is neat b/c transactions didn't need to lock!
  so they can run quickly from client caches
  just one msg exchange w/ validator per transaction
    not one locking exchange per record used
  OCC excellent for T2 which didn't conflict with anything
    we got lucky for T1 T3 T4, which do conflict

Validation example 2 -- sometimes must abort:
  initially, x=0 y=0
  T1: Rx0 Wx1
  T2: Rx0 Wy1
  T3: Ry0 Rx1
  values not consistent w/ any serial order!
    T1 -> T3 (via x)
    T3 -> T2 (via y)
    T2 -> T1 (via x)
    there's a cycle, so not the same as any serial execution
  perhaps T3 read a stale y=0 from cache
    or T2 read a style x=0 from cache
  in this case validation can abort one of them
    then others are OK to commit
```

```
  e.g. abort T2
    then T1, T3 is OK (but not T3, T1)

How should client handle abort?
  roll back the program (including writes to program variables)
  re-run from start of transaction
  hopefully won't be conflicts the second time
  OCC is best when conflicts are uncommon!

Do we need to validate read-only transactions?
  example:
    initially x=0 y=0
    T1: Wx1
    T2: Rx1 Wy2
    T3: Ry2 Rx0
  i.e. T3 read a stale x=0 from its cache, hadn't yet seen invalidate.
  need to validate in order to abort T3.
  other OCC schemes can avoid validating read-only transactions
    keep multiple versions -- but Thor and my schemes don't

Is OCC better than locking?
  yes, if few conflicts
    avoids lock msgs, clients don't have to wait for locks
  no, if many conflicts
    OCC aborts, must re-start, perhaps many times
    locking waits
  example: simultaneous increment
    locking:
      T1: Rx0 Wx1
      T2: -------Rx1  Wx2
    OCC:
      T1: Rx0 Wx1
      T2: Rx0 Wx1
      fast but wrong; must abort one

We really want *distributed* OCC validation
  split storage and validation load over servers
  each storage server sees only xactions that use its data
  each storage server validates just its part of the xaction
  two-phase commit (2PC) to check that they all say "yes"
    only really commit if all relevant servers say "yes"

Can we just distribute validation scheme #1?
  imagine server S1 knows about x, server S2 knows about y
  example 2 again
    T1: Rx0 Wx1
    T2: Rx0 Wy1
    T3: Ry0 Rx1
  S1 validates just x information:
    T1: Rx0 Wx1
    T2: Rx0
    T3: Rx1
    answer is "yes" (T2 T1 T3)
  S2 validates just y information:
    T2: Wy1
    T3: Ry0
    answer is "yes" (T3 T2)
  but we know the real answer is "no"

So simple distributed validation does not work
  the validators must choose consistent orders!

Validation scheme #2
  Idea: client (or TC) chooses timestamp for committing xaction
    from loosely synchronized clocks, as in Thor
  validation checks that reads and writes are consistent with TS order
  solves distrib validation problem:
    timestamps tell the validators the order to check
    so "yes" votes will refer to the same order

Example 2 again, with timestamps:
  T1@100: Rx0 Wx1
```

```
    T2@110: Rx0 Wy1
    T3@105: Ry0 Rx1
    S1 validates just x information:
      T1@100: Rx0 Wx1
      T2@110: Rx0
      T3@105: Rx1
      timestamps say order must be T1, T3, T2
      does not validate! T2 should have seen x=1
    S2 validates just y information:
      T2@110: Wy1
      T3@105: Ry0
      timstamps say order must be T3, T2
      validates!
    S1 says no, S2 says yes, two-phase commit coordinator will abort

  What have we given up by requiring timestamp order?
    example:
      T1@100: Rx0 Wx1
      T2@50: Rx1 Wx2
    T2 follows T1 in real time, and sees T1's write
    but T2 will abort, since TS says T2 comes first, so T1 should have seen x=2
      could have committed, since T1 then T2 works
    this will happen if client clocks are too far off
      if T1's client clock is ahead, or T2's behind
    so: requiring TS order can abort unnecessarily
      b/c validation no longer *searching* for an order that works
      instead merely *checking* that TS order consistent w/ reads, writes
      we've given up some optimism by requiring TS order
    maybe not a problem if clocks closely synched
    maybe not a problem if conflicts are rare

  Problem with schemes so far:
    commit messages contained *values*, which can be big
    could instead use version numbers to check whether
      later xaction read earlier xaction's write
    let's use writing xaction's TS as record version number

  Validation scheme #4
    tag each DB record (and cached record) with TS of xation that last wrote it
    validation requests carry TS of each record read

  Our example for scheme #4:
    all values start with timestamp 0
    T1@100: Rx@0 Wx
    T2@110: Rx@0 Wy
    T3@105: Ry@0 Rx@100
    note:
      reads have timestamp that was in read record, not value
      writes don't include either value or timestamp
    S1 validates just x information:
      orders the transactions by timestamp:
      T1@100: Rx@0 Wx
      T3@105: Rx@100
      T2@110: Rx@0
      the question: does each read see the most recent write?
        T3 is ok, but T2 is not
    S2 validates just y information:
      again, sort by TS, check each read saw latest write:
      T3@105: Ry@0
      T2@110: Wy
      this does validate
    so scheme #4 abort, correctly, reasoning only about version TSs

  what have we give up by thinking about version #s rather than values?
    maybe version numbers are different but values are the same
    e.g.
      T1@100: Wx1
      T2@110: Wx2
      T3@120: Wx1
      T4@130: Rx1@100
    timestamps say we should abort T4 b/c read a stale version
      should have read T3's write
```

```
    so scheme #4 will abort
  but T4 read the correct value -- x=1
    so abort wasn't necessary

Problem: per-record timestamp might use too much storage space
  Thor wants to avoid space overhead
  maybe important, maybe not

Validation scheme #5
  Thor's invalidation scheme: no timestamps on records
  how can validation detect that a transaction read stale data?
  it read stale data b/c earlier xaction's invalidation hadn't yet arrived!
  so server can track invalidation msgs that might not have arrived yet
    "invalid set" -- one per client
    delete invalid set entry when client ACKs invalidation msg
    server aborts committing xaction if it read record in client's invalid set
    client aborts running xaction if it read record mentioned in invalidation

Example use of invalid set
  [timeline]
  Client C1:
    T2@105 ... Rx ... 2PC commit point
    imagine that client acts as 2PC coordinator
  Server:
    VQ: T1@100 Wx
    T1 committed, x in C1's invalid set
      server has sent invalidation message to C1

Three cases:
  1. invalidation arrives before T2 reads
     Rx will miss in client cache, read from data from server
     client will (probably) return ACK before T2 commits
     server won't abort T2
  2. invalidation arrives after T2 reads, before commit point
     client will abort T2 in response to invalidation
  3. invalidation arrives after 2PC commit point
     i.e. after all servers replied to prepare
     this means the client was still in the invalid set when
       the server tried to validate the transaction
     so the server aborted, so the client will abort too
  so: Thor's validation detects stale reads w/o timestamp on each record

Performance

Look at Figure 5
  AOCC is Thor
  comparing to ACBL: client talks to srvr to get write-locks,
   and to commit non-r/o xactions, but can cache read locks along with data
  why does Thor (AOCC) have higher throughput?
    fewer msgs; commit only, no lock msgs
  why does Thor throughput go up for a while w/ more clients?
    apparently a single client can't keep all resources busy
    maybe due to network RTT?
    maybe due to client processing time? or think time?
    more clients -> more parallel xactions -> more completed
  why does Thor throughput level off?
    maybe 15 clients is enough to saturate server disk or CPU
    abt 100 xactions/second, about right for writing disk
  why does Thor throughput *drop* with many clients?
    more clients means more concurrent xactions at any given time
    more concurrency means more chance of conflict
    for OCC, more conflict means more aborts, so more wasted CPU

Conclusions
  fast client caching + transactions would be excellent
  distributed OCC very interesting, still an open research area
  avoiding per-record version #s doesn't seem compelling
  Thor's use of time was influential, e.g. Spanner
```

6.824 2015 Lecture 22: Peer to peer system

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

P2P systems

- Today: look at peer-to-peer (P2P) systems like Bittorrent and Chord
- classic implementation of file sharing services: users talk to a centralized server to download file
- it should be possible for users to talk to each other to get files
- the peer-to-peer dream: no centralized components, just built out of people's computers

Why P2P?

- **(+)** spreads the work of serving the files over a huge # of PCs
- **(+)** might be easier to deploy than a centralized system
    - no one has to buy a centralized server with a lot of bandwidth and storage
- **(+)** if you play your card rights, the # of resources should scale naturally with the # of users => less chance of overload
- **(+)** no centralized server => less chance to fail (harder to attack with DoS)
- `=>` so many advantages! Why does anyone build non-P2P systems?
- **(-)** it takes a sophisticated design to lookup a file in a system of a million users `<=>` finding stuff is hard (can't just ask a DB)
- **(-)** user computers are not as reliable as servers in a datacenter. users take their computers offline, etc.
- **(-)** some P2P systems are open (BitTorrent) but some are closed, where only say Amazon's computer are participating in this scheme (sort of like Dynamo).
    - `=>` in open systems, there will be malicious users `=>` easy to attack

The result is that P2P software has certain niches that you find it in

- systems where there's a lot of data, like online video services
- chat systems
- in settings where having a central server is not *natural* like Bitcoin
    - it would be nice for DNS to be decentralized for instance
- *seems* like the dominant use has been to serve illegal files

BitTorrent

Pre-DHT BitTorrent

Diagram:

```
 ---        ---
| W |      | T |
 ---        ---
/\         /\
| click on /  .torrent file
\         /
 ---/----        ---
| C |  <------> | C |
 ---             ---
```

- client goes to a webserver and downloads a .torrent file
- torrent file stores the hash of the data and the address of a tracker
- the tracker keeps a record of all the clients who have downloaded this file and maybe still have it and maybe would be willing to serve it to other clients
- client contacts tracker, asks about other clients
- the huge gain here is that the file is transfered between the clients directly, and the webserver and tracker aren't incurring too much cost

DHT-based BitTorrent (Trackerless torrents)

- tracker is a single point of failure
- can fix this by replicating them or having extra trackers
- the users who download the file also form a giant distributed key value store (a DHT)
- clients still have to talk to the original web server to get the `infohash` of the file it wants to download
- it uses it as a key to do a lookup in the DHT
- the values are IP addresses of other clients who have the files
- this really replaces the tracker
- how to find an entry node in the DHT? Maybe the client has some well known nodes hardcoded
- *maybe* the DHT is more reliable and less vulnerable to legal (subpoena) and technical attacks
- BitTorrent is fantastically popular: tens of millions of users `=>` giant DHT, however, most torrents are backed by real trackers

How did DHTs start?

- a bunch of research 15 years ago on how to build DHTs
- point was to harness the millions of computers on the Internet to provide something that was close to a database
- the interface is very simple: `Put(k, v), Get(k)`
  - hope is that puts are reflected in gets (after a while)
- in practice it is hard to build consistent systems
- little guarantees about availability of data and consistency
  - system does not guarantee to keep your data when you do a `Put()`
- still difficult to build, even with these weak guarantees

DHT designs

1. Flood everyone with Get's when you want to get a key
   - `=>` system can't handle too much load
2. Suppose everyone agreed to the whole list of nodes in the DHT.
   - Then you can have some hashing convention that hashes a key to an exact node and lookups are efficient.
   - Critical that all agree otherwise A sends put to X and B sends get to Y for the same key `k`
   - The real problem is that it's hard to keep tables up to date and accurate.

What we want:

- We're looking for a system where each node only has to know about a few other nodes.
- We don't want the node to send too many messages to do a lookup
- the rough approach that all DHT take is to define a global data structure that is layered across nodes
- Bad idea: organize all nodes as a binary tree, data is stored in leaf nodes such that lower keys are in the left most nodes
  - all traffic goes through root (bad) => if root goes down, partition
  - how can we replace nodes that go down?

Chord

- numbers in a circular ID space (like integers modulo p) from 0 to 2^160 - 1
- each node picks a random ID in this space as its node ID
- the keys have identifiers in this space, and we want the identifiers to have a uniform distribution, because we use it to map the key to a node identifier `=>` use a hash on the actual keys to get their identifier
- the node responsible for a key is the first *closest* node to that key in a clockwise

direction: known as its **successor**
- closeness `= |node ID - key ID|`

Slow but correct scheme:

- through some sort of hocus-pocus, every node just has to know about its own successor (say, node 2's successor is node 18, etc)
- we can always do a lookup starting at every node simply by following these successor pointers
  - this is called *routing*
  - all about forwarding a lookup message to a node further one the ring
- this is slow, with millions of nodes could be sending millions of messages for a single lookup
- need time logarithmic in the total # of nodes in the DHT `=>` each hop has to be able to compute the distance between it and any target key
- in Chord, every node has a *finger table* of 160 entries
- the finger table of node `n` has entry `i`:
  - `f[i] = successor(n + 2^i)`
  - `=>` the 159th entry will point to some node `n + 2^159` roughly halfway across the ID space
- each hop is on the order of 50 milliseconds, if the hops are halfway around the world
  - `=>` around 1 second to go through 20 nodes `=>` some applications might not take this well (BitTorrent is okay, because it only stores IPs in the DHT)
- when nodes join, they get a copy of the entry node's fingerprint
  - not accurate for the new node, but good enough
  - `=>` have to correct the table `=>` for the `i` th entry do a lookup for `n+2^i` and set `f[i]` to the address of the node that replied
- every lookup is likely to encounter a dead node and timeouts take a long time
- the churn in BitTorrent is quite high
  - 10 million people participate this hour, the next hour there will be other 10 million people `=>` hard to keep finger table updated
- `log n` lookup time is not that great
- finger tables are only used to speed up lookups
- each node must correctly know its successor for Chord to be correct
  - so that Gets by one node see the Puts by another node
- when a node first joins, it does a lookup on its own identifier to find its successor
  - `--> 10 -- [new node 15] --> 20 -->`
  - 15 sets its successor pointer to 20
  - so far no one is doing lookups
  - 15 isn't really part of the system until 10 knows about it
  - every node periodically asks its successor who they think their predecessor is

- `10: hey 20, who's your predecessor?`
- `20: my predecessor is 15`
- `10: oh, thought it was me, so let me set 15 as my successor then`
- `15: oh, hi 10, thanks for adding me as your sucessor, let me add you as my pr`
- this is called stabilization

Example:

```
10 -> 20

12, 18 join

10 -> 20, 12->20, 18->20, 20->18

10 -> 18, 18->10, 18->20, 20->18, 12 ->18

10 -> 18,18->20, 20->18, 12 ->18, 18->12

10 -> 12, 12->10 12->18, 18->12, 18->20, 20->18
```

- when a node gets a closer predecessor, it transfers keys that would be closer to its predecessor there

If nodes fail, can we do lookups correctly?

- suppose an intermediate node fails in the lookup procedure, then the initiating node can simply pick another
- towards the end of the lookup process, finger tables are not used anymore. instead successor pointers are followed `=>` if a node fails then the lookup cannot proceed `=>` nodes must remember successors of their successors to be able to proceed
- the probability of a partition occurring is low on the Internet

6.824 2015 original notes

```
Lecture outline:
  peer-to-peer (P2P)
  BitTorrent
  DHTs
  Chord

Peer-to-peer
  [user computers, files, direct xfers]
  users computers talk directly to each other to implement service
    in contrast to user computers talking to central servers
  could be closed or open
  examples:
    skype, video and music players, file sharing

Why might P2P be a win?
  spreads network/caching costs over users
  absence of server may mean:
    easier to deploy
    less chance of overload
    single failure won't wreck the whole system
    harder to attack

Why don't all Internet services use P2P?
```

```
  can be hard to find data items over millions of users
  user computers not as reliable than managed servers
  if open, can be attacked via evil participants

The result is that P2P has some successful niches:
  Client-client video/music, where serving costs are high
  Chat (user to user anyway; privacy and control)
  Popular data but owning organization has no money
  No natural single owner or controller (Bitcoin)
  Illegal file sharing

Example: classic BitTorrent
  a cooperative download system, very popular!
  user clicks on download link for e.g. latest Linux kernel distribution
    gets torrent file w/ content hash and IP address of tracker
  user's BT client talks to tracker
    tracker tells it list of other user clients w/ downloaded file
  user't BT client talks to one or more client's w/ the file
  user's BT client tells tracker it has a copy now too
  user's BT client serves the file to others for a while
  the point:
    provides huge download b/w w/o expensive server/link

BitTorrent can also use a DHT instead of / as well as a tracker
  this is the topic of today's readings
  BT clients cooperatively implement a giant key/value store
  "distributed hash table"
  the key is the file content hash ("infohash")
  the value is the IP address of a client willing to serve the file
    Kademlia can store multiple values for a key
  client does get(infohash) to find other clients willing to serve
    and put(infohash, self) to register itself as willing to serve
  client also joins the DHT to help implement it

Why might the DHT be a win for BitTorrent?
  single giant tracker, less fragmented than many trackers
    so clients more likely to find each other
  maybe a classic tracker too exposed to legal &c attacks
  it's not clear that BitTorrent depends heavily on the DHT
    mostly a backup for classic trackers?

How do DHTs work?

Scalable DHT lookup:
  Key/value store spread over millions of nodes
  Typical DHT interface:
    put(key, value)
    get(key) -> value
  loose consistency; likely that get(k) sees put(k), but no guarantee
  loose guarantees about keeping data alive

Why is it hard?
  Millions of participating nodes
  Could broadcast/flood request -- but too many messages
  Every node could know about every other node
    Then hashing is easy
    But keeping a million-node table up to date is hard
  We want modest state, and modest number of messages/lookup

Basic idea
  Impose a data structure (e.g. tree) over the nodes
    Each node has references to only a few other nodes
  Lookups traverse the data structure -- "routing"
    I.e. hop from node to node
  DHT should route get() to same node as previous put()

Example: The "Chord" peer-to-peer lookup system
  By Stoica, Morris, Karger, Kaashoek and Balakrishnan; 2001

Chord's ID-space topology
  Ring: All IDs are 160-bit numbers, viewed in a ring.
  Each node has an ID, randomly chosen
```

```
Assignment of key IDs to node IDs?
  Key stored on first node whose ID is equal to or greater than key ID.
    Closeness is defined as the "clockwise distance"
  If node and key IDs are uniform, we get reasonable load balance.
  So keys IDs should be hashes (e.g. bittorrent infohash)

Basic routing -- correct but slow
  Query is at some node.
  Node needs to forward the query to a node "closer" to key.
    If we keep moving query closer, eventually we'll win.
  Each node knows its "successor" on the ring.
    n.lookup(k):
      if n < k <= n.successor
        return n.successor
      else
        forward to n.successor
  I.e. forward query in a clockwise direction until done
  n.successor must be correct!
    otherwise we may skip over the responsible node
    and get(k) won't see data inserted by put(k)

Forwarding through successor is slow
  Data structure is a linked list: O(n)
  Can we make it more like a binary search?
    Need to be able to halve the distance at each step.

log(n) "finger table" routing:
  Keep track of nodes exponentially further away:
    New state: f[i] contains successor of n + 2^i
    n.lookup(k):
      if n < k <= n.successor:
        return successor
      else:
        n' = closest_preceding_node(k) -- in f[]
        forward to n'

for a six-bit system, maybe node 8's looks like this:
  0: 14
  1: 14
  2: 14
  3: 21
  4: 32
  5: 42

Why do lookups now take log(n) hops?
  One of the fingers must take you roughly half-way to target

There's a binary lookup tree rooted at every node
  Threaded through other nodes' finger tables
  This is *better* than simply arranging the nodes in a single tree
    Every node acts as a root, so there's no root hotspot
    But a lot more state in total

Is log(n) fast or slow?
  For a million nodes it's 20 hops.
  If each hop takes 50 ms, lookups take a second.
  If each hop has 10% chance of failure, it's a couple of timeouts.
  So in practice log(n) is better than O(n) but not great.

How does a new node acquire correct tables?
  General approach:
    Assume system starts out w/ correct routing tables.
    Use routing tables to help the new node find information.
    Add new node in a way that maintains correctness.
  New node m:
    Sends a lookup for its own key, to any existing node.
      This yields m.successor
    m asks its successor for its entire finger table.
  At this point the new node can forward queries correctly
  Tweaks its own finger table in background
    By looking up each m + 2^i
```

```
Does routing *to* new node m now work?
  If m doesn't do anything,
    lookup will go to where it would have gone before m joined.
    I.e. to m's predecessor.
    Which will return its n.successor -- which is not m.
  So, for correctness, m's predecessor needs to set successor to m.
    Each node keeps track of its current predecessor.
    When m joins, tells its successor that its predecessor has changed.
    Periodically ask your successor who its predecessor is:
      If that node is closer to you, switch to that guy.
    So if we have x m y
      x.successor will be y (now incorrect)
      y.predecessor will be m
      x will ask its x.successor for predecessor
        x learns about m
        sets x.successor to m
        tells m "x is your predecessor"
        called "stabilization"
  Correct successors are sufficient for correct lookups!

What about concurrent joins?
  Two new nodes with very close ids, might have same successor.
  Example:
    Initially 40 then 70
    50 and 60 join concurrently
    at first 40, 50, and 60 think their successor is 70!
    which means lookups for e.g. 45 will yield 70, not 50
    after one stabilization, 40 and 50 will learn about 60
    then 40 will learn about 50

To maintain log(n) lookups as nodes join,
  Every one periodically looks up each finger (each n + 2^i)

Chord's routing is conceptually similar to Kademlia's
  Finger table similar to bucket levels
    Both halve the metric distance for each step
    Both are about speed and can be imprecise
  n.successor similar to Kademlia's requirement that
    each node know of all the nodes that are very close in xor-space
    in both cases care is needed to ensure that different lookups
      for same key converge on exactly the same node

What about node failures?
  Assume nodes fail w/o warning. Strictly harder than graceful departure.
  Two issues:
    Other nodes' routing tables refer to dead node.
    Dead node's predecessor has no successor.
  If you try to route via dead node, detect timeout, treat as empty table entry.
    I.e. route to numerically closer entry instead.
  For dead successor
    Failed node might have been just before key ID!
      So we need to know what its n.successor was
    Maintain a _list_ of successors: r successors.
    Lookup answer is first live successor >= key
      or forward to *any* successor < key

Kademlia has a faster plan for this
  send alpha (or k) lookup RPCs in parallel, to different nodes
  send more lookups as previous ones return info about nodes closer to key
  single non-responsive node won't cause lookup to suffer a timeout

Dealing with unreachable nodes during routing is extremely important
  "Churn" is very high in open p2p networks
  People close their laptops, move WiFi APs, &c pretty often
  Measurement of Bittorrent/Kademlia suggest lookups are not very fast

Geographical/network locality -- reducing lookup time
  Lookup takes log(n) messages.
    But they are to random nodes on the Internet!
    Will often be very far away.
  Can we route through nodes close to us on underlying network?
```

```
  This boils down to whether we have choices:
    If multiple correct next hops, we can try to choose closest.

Idea:
  to fill a finger table entry, collect multiple nodes near n+2^i on ring
  perhaps by asking successor to n+2^i for its r successors
  use lowest-ping one as i'th finger table entry

What's the effect?
  Individual hops are lower latency.
  But less and less choice (lower node density) as you get close in ID space.
  So last few hops likely to be very long.
  Though if you are reading, and any replica will do,
    you still have choice even at the end.

What about security?
  Self-authenticating data, e.g. key = SHA1(value)
    So DHT node can't forge data
    Of course it's annoying to have immutable data...
  Can someone cause millions of made-up hosts to join?
    They don't exist, so routing will break?
    Don't believe new node unless it responds to ping, w/ random token.
  Can a DHT node claim that data doesn't exist?
    Yes, though perhaps you can check other replicas
  Can a host join w/ IDs chosen to sit under every replica?
    Or "join" many times, so it is most of the DHT nodes?
    Maybe you can require (and check) that node ID = SHA1(IP address)

Why not just keep complete routing tables?
  So you can always route in one hop?
  Danger in large systems: timeouts or cost of keeping tables up to date.

How to manage data?
  Here is the most popular plan.
  DHT doesn't guarantee durable storage
    So whoever inserted must re-insert periodically if they care
    May want to automatically expire if data goes stale (bittorrent)
  DHT does replicate each key/value item
    On the nodes with IDs closest to the key, where looks will find them
    Replication can help spread lookup load as well as tolerate faults
  When a node joins:
    successor moves some keys to it
  When a node fails:
    successor probably already has a replica
    but r'th successor now needs a copy

Retrospective
  DHTs seem very promising for finding data in large p2p systems
    Decentralization seems good for load, fault tolerance
  But: the security problems are difficult
  But: churn is a serious problem, particularly if log(n) is big
  So DHTs have not had the impact that many hoped for
```

6.824 2015 Lecture 23: Bitcoin

Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015.

Bitcoin

- an electronic currency system
- has a technical side and a financial, economic, social side
- maybe the 1st thing to ask: is it trying to do something better? is there a problem it solves for us?
- online payments use credit cards, why not just use them?
  - Pluses:
    - They work online
    - Hard for people to steal my credit card (there are laws about how credit card companies work so that if your number is stolen, you are protected)
  - Good/Bad:
    - Customer service # on the back allows you to reverse charges
      - this can prevent or create fraud
    - tied to some country's currency
  - Minuses
    - No way for me as a customer or a merchant to independently verify anything about a credit card transaction: do you have money, is the CC # valid?
      - it can be good if you don't want people finding out how much money you have
    - relies on 3rd parties: great way to charge fees on everything
    - 3% fees
    - settling time is quite long (merchants are not sure they are getting their money until after one month)
    - pretty hard to become a credit card merchant
      - credit card companies take a lot of risk by sending money to merchants who might not send products to customers, resulting in the credit card company having to refund customers
- For Bitcoin:
  - no 3rd parties are needed (well, not really true anymore)
  - fees are much smaller than 3%
  - the settling time is maybe 10 minutes
  - anyone can become a merchant

- Bitcoin makes the sequence of transactions verifiable by everyone and agree on it `=>` no need to rely on 3rd parties

OneBit

- simple electronic money system
- it has one server called OneBank
- each user owns some coins

Design:

```
OneBank server
```

- onebit xction:
    1. public key of new owner
    2. a hash of the last transfer record of this coin
    3. a signature done over this record by the private key of last owner
- bank keeps the list of transactions for each coin
- `x` transfer the coin to `y`
- `[T7: from=x, to=y; hash=h(prev tx); sig_x(this)]`
- `y` transfers the coin to `z`, gets a hamburger from McDonalds
- `[T8: from y, to=z; hash=h(T7); sig_y(this)]`
- what can go wrong?
    - if someone transfers a coin to `z` it seems very unlikely that anyone else other than `z` can spend that coin: because no one else can sign a new transaction with that coin since they don't have `z`'s private key
- we have to trust one bank to not let users double spend money
    - `y` can also buy a milkshake from Burger King with that same coin if the bank helps him
    - `[T8': from y, to=q'; hash=h(T7); sig_y(this)]`
    - the bank can show T8 to McDonalds and T8' to Burget King
    - (I love free food!)
    - as long as McDonalds and Burger King don't talk to each other and verify the transaction chain, they won't detect it

Bitcoin block chain

- bitcoin has a single block chain
- many server: more or less replicas, have copy of entire block chain
- each block in the block chain looks like this:

    - hash of previous block
    - set of transactions
    - nonce
    - current time
- xactions have two stages
    - first it is created and sent out to the network
    - then the transaction is incorporated into the block chain

How are blocks created? Mining

All of the peers in the bitcoin network try to create the next block:

- each peer takes all transactions that have arrived since the previous block was created and try to append a new block with them
- the rules say that a hash of a block has to be less than a certain number (i.e. it has a # of leading of zeros, making it hard to find)
- each of the bitcoin peers adjust the `nonce` field in the block until they get a hash with a certain # of leading zeros
- the point of this is to make it expensive to create new blocks
    - for a single computer it might take months to find such a nonce
- the # of leading zeros is adjusted so that on average it takes 10 minutes for a new block to be added
    - clients monitor the `currentTime` field in the last 5 transactions or so and if they took to little time, they add another zero to # of target zeros
        - everyone obeys the protocol because if they don't the others will either reject their block (say if it has the wrong # of zeros or a wrong timestamp)

The empty block chain

- "In the beginning there was nothing, and then Satoshi created the first block."
- "And then people started mining additional blocks, with no transactions."
- "And then they got mining reward for each mined block."
- "And that's how users got Bitcoins."
- "And then they started doing transactions."
- "And then there was light."

What does it take to double spend

If a tx is in the block chain, can the system double spend its coins?

- forking the block chain is the only way to do this

- can the forks be hidden for long?
- if forks happens, miners will pick either one and continue mining
- when a fork gets longer, everyone switches to it
    - if they stay on the shorter fork, they are likely to be outmined by the others and waste work, so they will have incentive to go on the longer one
    - the tx's on the shorter fork get incorporated in the longer one
    - committed tx's can get undone => people usually wait for a few extra blocks to be created after a tx's block
- this is where the 51% rule comes in: if 51% of the computing power is honest the protocol works correctly
- if more than 51% are dishonest, then they'll likely succeed in mining anything they want
- probably the most clever thing about bitcoin: as long as you believe than more than half the computing power is not cheating, you can be sure there's no double spending

Good and bad parts of design

- (+) publicly verifiable log
- (-) tied to a new currency and it is very volatile
    - lots of people don't use it for this reason
- (+/-) mining-decentralized trust

Hard to say what will happen:

- we could be all using it in 30 years
- or, banks could catch up, and come up with their own verifiable log design